AD-A128 232 DIANA REFERENCE MANUAL REVISION 3(U) TARTAN LABS INC 1/3
PITTSBURGH PA A EVANS ET AL. 28 FEB 83 TL-83-4
MDA903-82-C-0148

UNCLASSIFIED F/G 9/2 NL

(10)

AD A128232

DTIC FILE COPY

TARTAN LABORATORIES INCORPORATED

DIANA REFERENCE MANUAL

Revision 3

Arthur Evans Jr.
Kenneth J. Butler
Tartan Laboratories Incorporated
Editors, Revised Diana Reference Manual

G. Goos
Institut fuer Informatik II, Universitaet Karlsruhe
Wm. A. Wulf
Carnegie-Mellon University
Editors, Original Diana Manual

Prepared for
Ada Joint Program Office
801 North Randolph Street
Arlington Virginia 22203
Contract Number MDA903-82-C-0148

Prepared by
Tartan Laboratories Incorporated
477 Melwood Avenue
Pittsburgh PA 15213

1983 February 28

TL 83-4

Diana is being maintained and revised by Tartan Laboratories Inc. for the Ada Joint Program Office of the Department of Defense under contract number MDA903-82-C-0148 (expiration date: 1983 February 28). The Project Director of Diana Maintenance for Tartan Laboratories is Arthur Evans, Jr.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TL 83-4	2. GOVT ACCESSION NO. AD-A128232	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DIANA REFERENCE MANUAL, Revision 3		5. TYPE OF REPORT & PERIOD COVERED Contract deliverable 0002AC
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Arthur Evans Jr., Kenneth J. Butler, editors		8. CONTRACT OR GRANT NUMBER(s) MDA903-82-C-0148
9. PERFORMING ORGANIZATION NAME AND ADDRESS Tartan Laboratories Inc. 477 Melwood Ave. Pittsburgh PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 801 North Randolph Street Arlington VA 22203		12. REPORT DATE 1983 Feb 28
		13. NUMBER OF PAGES viii + 201
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) DCASMA Pittsburgh 1610-S Federal Building 1000 Liberty Avenue Pittsburgh PA 15222		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Diana, Ada, compiler, programming language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document describes Diana, a Descriptive Intermediate Attributed Notation for Ada, being both an introduction and reference manual for it. Diana is an abstract data type such that each object of the type is a representation of an intermediate form of an Ada program. Although the initial uses of this form were for communication between the Front and Back Ends of an Ada compiler, it is also intended to be suitable for use with other tools in an Ada programming environment. Diana resulted from a merger of the best properties of two earlier similar intermediate forms: TCOL and AIDA.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This document describes *Diana*, a Descriptive Intermediate Attributed Notation for ADA, being both an introduction and reference manual for it. DIANA is an abstract data type such that each object of the type is a representation of an intermediate form of an ADA program. Although the initial uses of this form were for communication between the Front and Back Ends of an ADA compiler, it is also intended to be suitable for use with other tools in an ADA programming environment.

DIANA resulted from a merger of the best properties of two earlier similar intermediate forms: TCOL and AIDA.

ACKNOWLEDGMENTS

ACKNOWLEDGMENTS FOR THE FIRST EDITION

DIANA is based on two earlier proposals for intermediate forms for ADA programs: TCOL and AIDA. It could not have been designed without the efforts of the two groups that designed these previous schemes. Thus we are deeply grateful to:

- AIDA: Manfred Dausmann, Guido Persch, Sophia Drossopoulou, Gerhard Goos, and Georg Winterstein—all from the University of Karlsruhe.
- TCOL: Benjamin Brosgol (Intermetrics), Joseph Newcomer (Carnegie-Mellon University), David Lamb (CMU), David Levine (Intermetrics), Mary Van Deusen (Prime), and Wm. Wulf (CMU).

The actual design of DIANA was conducted by teams from Karlsruhe, Carnegie-Mellon, Intermetrics and Softech. Those involved were Benjamin Brosgol, Manfred Dausmann, Gerhard Goos, David Lamb, John Nestor, Richard Simpson, Michael Tighe, Larry Weissman, Georg Winterstein, and Wm. Wulf. Assistance in creation of the document was provided by Jeff Baird, Dan Johnston, Paul Knueven, Glenn Marcy, and Aaron Wohl—all from CMU.

We are grateful for the encouragement and support provided for this effort by Horst Clausen (IABG), Larry Druffel (DARPA), and Marty Wolfe (CENTACS), as well as our various funding agencies.

Finally, the design of DIANA was conducted at Eglin Air Force Base with substantial support from Lt. Col. W. Whitaker. We could not have done it without his aid.

DIANA's original design was funded by Defense Advanced Research Projects Agency (DARPA), the Air Force Avionics Laboratory, the Department of the Army, Communication Research and Development Command, and the Bundesamt fuer Wehrtechnik und Beschaffung.

Gerhard Goos
Wm. A. Wulf
Editors, First Edition

ACKNOWLEDGMENTS FOR THIS EDITION

Subsequent to DIANA's original design, the ADA Joint Program Office of the United States Department of Defense has supported at TARTAN Laboratories Incorporated a continuing effort at revision. This revision has been performed by Arthur Evans, Jr., and Kenneth J. Butler, with considerable assistance from John R. Nestor and Wm. A. Wulf, all of TARTAN.

We are grateful to the following for their many useful comments and suggestions.

- Georg Winterstein, Manfred Dausmann, Sophia Drossopoulou, Guido Persch, and Juergen Uhl, all of the Karlsruhe ADA Implementation Group;
- Julie Sussman and Rich Shapiro of Bolt Beranek and Newman Inc.; and to
- Charles Wetherell and Peggy Quinn of Bell Telephone Laboratories

Additional comments and suggestions have been offered by Grady Booch, Benjamin Brosgol, Gil Hanson, Jeremy Holden, Bernd Krieg-Brueckner, David Lamb, H.-H. Nagell, Teri Payton, and Richard Simpson.

We thank the ADA Joint Program Office (AJPO) for supporting DIANA's revision, and in particular Lt. Colonel Larry Druffel, the director of AJPO. Valuable assistance as Contracting Officer's Technical Representative was provided first by Lt. Commander Jack Kramer and later by Lt. Commander Brian Schaar; we are pleased to acknowledge them.

DIANA is being maintained and revised by TARTAN Laboratories Inc. for the ADA Joint Program Office of the Department of Defense under contract number MDA903-82-C-0148 (expiration date: 1983 February 28). The Project Director of DIANA Maintenance for TARTAN is Arthur Evans, Jr.

PREFACE

PREFACE TO THE FIRST EDITION

This document defines *Diana*, an intermediate form of ADA [7] programs that is especially suitable for communication between the front and Back Ends of ADA compilers. It is based on the formal definition of ADA [6] and resulted from the merger of the best aspects of two previous proposals: AIDA [4, 10] and TCOL [2]. Although DIANA is primarily intended as an interface between the parts of a compiler, it is also suitable for other programming support tools and carefully retains the structure of the original source program.

The definition of DIANA *given here* is *expressed* in another notation, IDL, that is formally defined in a separate document [9]. The present document is, however, completely self-contained; those aspects of IDL that are needed for the DIANA definition are informally described before they are used. Interested readers should consult the IDL formal description *either* if they are concerned with a more precise definition of the notation *or* if they need to define other data structures in an ADA support environment. In particular, implementors may need to extend DIANA in various ways for use with the tools in a specific environment, and the IDL document provides information on how this may be done.

This version of DIANA has been "frozen" to meet the needs of several groups who require a stable definition in a very short timeframe. We invite comments and criticisms for a longer-term review. We expect to re-evaluate DIANA after some practical experience with using it has been accumulated.

PREFACE TO THIS EDITION

Since first publication of the DIANA Reference Manual in March, 1981, further developments in connection with ADA and DIANA have required revision of DIANA. These developments include the following:

- The original DIANA design was based on ADA as defined in the July 1980 ADA Language Reference Manual [7], referred to hereafter as ADA-80: the present revision is based on ADA as defined in the July 1982 ADA LRM [8], referred to hereafter as ADA-82.
- Experience with use of DIANA has revealed errors and flaws in the original design: these have been corrected.

This publication reflects our best efforts to cope with the conflicting pressures on us both to impact minimally on existing implementations and to create a logically defensible design.

TARTAN Laboratories Inc. invites any further comments and criticisms on DIANA in general, and this version of the reference manual in particular. Any correspondence may be sent *via* ARPANet mail to DIANA-QUERY@USC-ECLB. Paper mail may be sent to

DIANA Manual
TARTAN Laboratories Inc.
477 Melwood Avenue
Pittsburgh PA 15213

We believe the changes made to DIANA make no undue constraint on any DIANA users or potential DIANA users, and we wish to hear from those who perceive any of these changes to be a problem.

CHAPTER 1
INTRODUCTION

The purpose of standardization is to aid the creative craftsman, not to enforce the common mediocrity [11].

In a programming environment such as that envisioned for ADA[1], there will be a number of tools—formatters (pretty printers), language-oriented editors, cross-reference generators, test-case generators, and the like. In general, the input and output of these tools is *not* the source text of the program being developed; instead it is some intermediate form that has been produced by another tool in the environment. This document defines *Diana*, Descriptive Intermediate Attributed Notation for ADA. DIANA is an intermediate form of ADA programs which has been designed to be especially suitable for communication between two essential tools—the Front and Back Ends of a compiler—but also to be suitable for use by other tools in an ADA support environment. DIANA encodes the results of lexical, syntactic, and *static* semantic analysis, but it does *not* include the results of *dynamic* semantic analysis, of optimization, or of code generation.

It is common to refer to a scheme such as DIANA as an intermediate *representation* of programs. Discussions of DIANA, including those in this document, undoubtedly use this and similar terminology. Unfortunately, too often the word *representation* suggests a concrete realization such as a particular data structure in primary memory or on a file. It is important for the reader to keep in mind that DIANA does *not* imply either of these. Indeed, quite the opposite is the case: it was carefully defined to permit a wide variety of realizations as different concrete data or file structures.

A far more accurate characterization of DIANA is that it is an *abstract data type*. The DIANA representation of a particular ADA program is an instance of this abstract type. As with all abstract types, DIANA defines a set of operations that provide the only way in which instances of the type can be examined or modified. The actual data or file structures used to represent the type are

---

[1] Ada is a registered Trademark of the Ada Joint Program Office, Department of Defense, United States Government.

hidden by these operations, in the sense that the implementation of a private type in ADA is hidden.

We often refer to a DIANA 'tree', 'abstract syntax tree', or 'attributed parse tree'; similarly, we refer to 'nodes' in these trees. In the context of DIANA as an abstract data type, it is important to appreciate what *is* and *is not* implied by such terms. We are *not* saying that the data structure used to implement DIANA is necessarily a tree using pointers and the like. Rather, we are using the notion of attributed trees as the *abstract model* for the definition of DIANA.

An abstract data type consists of (a) a set of *values* (the *domain* of the type) together with (b) a set of *operations* on those values. The specification of an abstract type must define both its values and its operations. The *abstract modeling* method of specifying an abstract type provides these definitions by defining the values in terms of some mathematical entity with which the reader is presumed to be familiar; the operations of the type are then defined in terms of their effect on the modeling entities. In the case of DIANA, for example, the mathematical model is that of attributed trees. The reader should always bear in mind that the trees being discussed are merely conceptual ones; they are the *model* of the values in the DIANA domain. They may or may not exist as an explicit part of an implementation of the DIANA abstract type[2].

1.1. Design Principles

The design of DIANA is based on the collection of principles that are discussed in this section. As with any design intended for practical use, some compromise of these principles has on occasion been necessary. The frequency of deviations from the principles is extremely low, however, and an understanding of the principles will help the reader to understand DIANA.

Section 1.1.1 presents those principles that motivated the original design of DIANA, and Section 1.1.2 presents those principles that have governed changes made since. Section 1.1.3 defines what it means to be a DIANA *user* (*i.e.*, producer or consumer) and Section 1.1.4 presents a *lacuna* of the entire DIANA definition effort.

---

[2] An alternative definitional method, algebraic axioms, would have avoided explicit reference to a model such as trees and hence might have been less suggestive of an implementation. We chose not to use this method in order to retain a close correspondence between Diana and Ada's formal definition [6].

1.1.1. Original Design

The following principles governed the original design of DIANA:

- *Diana is representation independent*. As noted above, we strove to avoid implying any particular implementation strategy for the DIANA abstract type. For example, where implementation-specific information is needed in a DIANA representation (such as values on the target-machine), we make reference to other abstract types for representing these data; each implementation is expected to supply the implementation of these types. In addition, we strove to avoid any implications for the strategies to be used in implementing Front or Back Ends of compilers, or, for that matter, any other environment tools. Finally, we provide an explicit mechanism for implementations to extend or contract the DIANA form in a consistent manner to cater to implementation-specific purposes.

- *Diana is based on ADA's formal definition* [6], referred to hereafter as the *AFD*. In defining an intermediate representation of ADA, we face three problems: what is the representation of a particular program, what does that representation *mean* (*i.e.*, what is the semantics of the particular instance of the representational scheme), and when is the representation *consistent* (*i.e.*, meaningful)? Since the AFD already provides the latter two of these[3], we have chosen to stay as close as possible to the definitional scheme used there—particularly to the abstract syntax. Thus, in this document we can focus exclusively on the first of these questions, namely how particular programs are represented.

- *Regularity is a principal characteristic of Diana*. Regularity of description and notation was a principal goal. We believe that this regularity is an important aspect of both understanding *and processing* a DIANA intermediate form.

- *Diana must be efficiently implementable*. As noted above, DIANA is best viewed as an abstract data type. Its specification is more abstract than is directly supported by current programming languages, including ADA. Nonetheless, DIANA is intended to be used! Hence, it is essential that there exist an efficient implementation of it (or actually, several different efficient implementations of it) in contemporary languages—especially ADA itself. Later chapters deal with this issue explicitly; for now, the important point is that implementability was a primary consideration and that such implementations *do* exist.

- *Consideration of the kinds of processing to be done is paramount*. Although the primary purpose of DIANA is communication between the Front and Back Ends of compilers, other environment tools will use it

---

[3] Some problems with the AFD as an answer to these questions are addressed in Section 1.1.4 on page 14.

as well. The needs of such programs were considered carefully. They influenced a number of the DIANA design decisions, including the following:

- We define two trees—an Abstract Syntax Tree constructed prior to semantic analysis (see Appendix II), and an attributed tree (the DIANA structure) constructed as a result of static semantic analysis. These two structures are, of course, closely related. By defining both of them, we extend the applicability of DIANA to include those tools that need only the parsed form.

- We considered the *size* of (various implementations of) DIANA representations, and we made careful tradeoffs between this size and processing speed. We envision that at least some ADA support environments will be implemented on small computing systems; hence, we considered it essential that DIANA be usable on these systems.

- We *never* destroy the structure of the original source program; except for purely lexical issues (such as the placement of comments), it is always possible to regenerate the source text from its DIANA form. See Appendix III.

- We permit the possibility of extending the DIANA form to allow the inclusion of information for other kinds of processing. Of particular concern, for example, are extensions to encode information needed by various optimization and code-generation strategies.

- *In Diana, there is a single definition of each Ada entity.* Each definable entity, *e.g.* variable, subprogram, or type, is represented by a single defining occurrence in DIANA. Uses of the entity always[4] refer to this defining occurrence. Attributes at this definition point make it possible for all information about the entity to be determined. Thus, although the defining occurrences are part of the program tree, the set of them plays the same role as a dictionary or symbol table in conventional compiler terminology[5].

- *Diana must respond to the issues posed by Ada's separate compilation facility.* It is *not* in the domain of DIANA to provide the library management upon which separate compilation of ADA is based. Nonetheless, the possibility of separate compilation affects the design

---

[4] There is a single exception: Private types use a different defining occurrence for references inside the package body in which they are defined than they do elsewhere. See Section 3.5.1.2 on page 104.

[5] Note, in particular, that an implementation may wish to separate these defining occurrences so that, for example, they may be the *only* portion of the representation present for separately compiled units. Such implementations are completely consistent with the Diana philosophy. Other implementation options are discussed in Chapter 6.

of DIANA in two ways:

- The possibility of separate compilation places certain restrictions on DIANA and requires the possibility of certain indirect references. We take care, for example, never to require forward references to entities whose definition may be separately compiled.

- We recognize that many library systems may wish to store the DIANA form of a compilation unit—in order to support optimization across compilation units, for example. Various design decisions in DIANA were influenced by this possibility.

- *There must be at least one form of the Diana representation that can be communicated between computing systems.* We have defined in Chapter 5 an externally visible ASCII form of the DIANA representation of an ADA program. In this form, the DIANA representation can be communicated between arbitrary environment tools and even between arbitrary computing systems. The form may also be useful during the development of the environment tools themselves.

1.1.2. Principles Governing Changes

The design principles just listed that governed the original design of DIANA have been augmented during this phase of modification by additional principles. It is important that these, too, be documented.

- *Diana will be changed only when something is sufficiently wrong that it requires change.* We state this metric despite the fact it is such a broad characterization that deciding when something is 'sufficiently wrong' is clearly judgmental. Nonetheless, the principle has utility. For example, it implies that we not make cosmetic changes, no matter how obvious it might be that the change would result in a better product. Our motto: 'If it's not broken, don't fix it.'

- *We do not unduly impact existing Diana users.* Thus we refrain from changes whose impact on existing implementations significantly exceeds anticipated future benefits. Of course, changes with a large enough savings down the road may be made even if doing so affects current implementations. Again, there is a judgmental call here.

- *It is often necessary to make some decision.* In several cases, either of two or more ways to proceed has seemed equally plausible, and we have been unable to determine any significant advantage to any decision. Nonetheless, in such cases we have made a decision, since we judge a slightly incorrect decision to be better for the DIANA community than no decision. At least, there is a standard way for DIANA users to to proceed.

- *Where possible we have preserved the style of the original Diana*

design. Stylistic concerns include such issues as creating IDL classes for attributes, preserving the same naming conventions, and so on.

- *Diana does not unnecessarily deviate from ADA's formal definition.* Even though the formal definition effort apparently is no longer being actively pursued[6], we continue to adhere to its style.

Unfortunately the guidelines just presented and those of the previous section are sometimes in conflict. For example, consider a minor inconsistency found in the original design. The principle of *consistency* might suggest a change, while the principle of *sufficiently wrong* might suggest leaving it alone. What we have done is to be *reasonable* in considering changes. DIANA is intended to be used, and we continue to strive to keep DIANA responsive to the needs of its users.

1.1.3. What is a 'DIANA User'

Inasmuch as DIANA is an abstract data type, there is no need that it be implemented in any particular way. Additionally, because DIANA is extendable, a particular implementation may choose to use a superset of the DIANA defined in this DRM. In the face of innumerable variations on the same theme, we feel it is appropriate to offer a definition of what it means to *use* DIANA. Since it makes sense to consider DIANA only at the interfaces, it is appropriate to consider two types of DIANA users: those which *produce* DIANA, and those which *consume* it[7]. In addition, some implementations (particularly compilers) may claim to *employ* DIANA as an intermediate form, even though neither interface to external DIANA is provided. We consider these three aspects in turn:

producer In order for a program to be considered a DIANA *producer*, it must produce as output a structure that includes all of the information contained in DIANA as defined in this document. Every attribute defined herein must be present, and each attribute must have the value defined for correct DIANA and may not have any other value. This requirement means, for example, that additional values, such as the evaluation of non-static expressions, may not be represented using the DIANA-defined attributes. An implementation is not prevented from defining additional attributes, and in fact it is expected that most DIANA producers will

---

[6]The formal definition is based on Ada-80, and there is no visible intent to upgrade it to Ada-82.

[7]These are not mutually exclusive; for example, a compiler Front End that produces Diana may also read Diana for separate compilation purposes.

also produce additional attributes.

There is an additional requirement on a DIANA producer: The DIANA structure must have the property that it could have been produced from a legal ADA program. This requirement is likely to impinge most strongly on a tool other than a compiler Front End that produces DIANA. As an example of this requirement, in an arithmetic expression, an offspring of a multiplication could not be an addition but would instead have to be a parenthesized node whose offspring was the addition, since ADA's parsing rules require the parentheses. The motivation for this requirement is to ease the construction of a DIANA consumer, since the task of designing a consumer is completely open-ended unless it can make some reasonable assumptions about its input.

consumer In order for a program to be considered a DIANA *consumer*, it must depend on no more than DIANA as defined herein. This restriction does not prevent a consumer from being able to take advantage of additional attributes that may be defined in an implementation; however, the consumer must also be able to accept input that does not have these additional attributes. It is also incorrect for a program to expect attributes defined herein to have values that are not here specified. For example, it is wrong for a program to expect the attribute *sm_value* to contain values of expressions that are not static.

employer The definition of a DIANA *employer* is more difficult. The intent is that the intermediate form must be close to DIANA; the problem is that we have no useful metric for *close*. In addition, the lack of a visible external representation of the intermediate form apparently precludes application of any validation procedure. This point is addressed further below.

There are two attributes that are defined herein that are *not* required to be supported by a DIANA user: *lx_comments* and *lx_srcpos*. We believe that these attributes are too implementation specific to be required for all DIANA users.

It is instructive to examine the problems suggested above of defining a DIANA *employer*. Inasmuch as papers have begun to appear in the literature in which a given implementation claims 'to use DIANA' or 'to be DIANA-like', we feel that it is appropriate to offer a metric against which to judge such claims. Consider the following three candidates for such a metric:

- A representation can properly be called DIANA if it contains all the same information that DIANA contains.

- A representation can properly be called DIANA if one can provide a reader/writer for transforming between the representation and DIANA.

- A representation can properly be called DIANA if it provides a package equivalent to the one described in Chapter 4 for accessing and modifying the structure.

Although the first two definitions have a certain appeal, it is unfortunately true that neither of them is at all adequate, since a little thought reveals that the original ADA source text meets either requirement. One repair possibility is to attempt to tighten up the second definition by restricting the reader/writer to be `simple', in some sense, but defining that sense appears to require Solomonic wisdom.

The third definition also has appeal, though it is again hard to use as a metric if the external interface is not actually provided in a useful way.

It is our opinion that it is not proper to claim that a given implementation uses DIANA unless *either* it meets the following two criteria:

- It must be able to read and/or write (as appropriate) the external form of DIANA defined in Chapter 5 of this document.

- That DIANA must meet the requirements of a DIANA producer or consumer as specified in this section.

or it meets this criterion:

- The implementation provides a package equivalent to that described in Chapter 4.

We hope that writers of papers will give consideration to this discussion.

1.1.4. Specification of DIANA

An important problem faced by new users of DIANA is to determine, for any particular ADA construct, just what DIANA is to be produced from it. Although the DIANA specification in Chapter 2 specifies precisely what nodes must exist, which attributes each node must contain, and what *type* the value of each attribute must have, it often says very little about what *value* the attribute is to have.

This problem is addressed in this document in several ways. Often, comments appear in Chapter 2 specifying or suggesting the intended value. In addition, the lengthy discussion of design rationale in Chapter 3 presents much additional information. Unfortunately, still more help is needed, and a complete solution to the problem of providing such help is beyond the capability of this document. The remainder of this section is speculation about the form such help might take.

What is needed is is a formal way to determine, given an ADA source text and a DIANA structure purported to be a correct representation of the source, whether or not the DIANA is in fact correct. For example, suppose that α is some ADA text and that δ purports to be a DIANA representation of it. Needed is a predicate π such that

$$\pi(\alpha, \delta)$$

is true if, and only if, δ correctly represents α.

Ideally, the structure of π should be such that it is accessible to a human reader who requires help in designing an ADA front end or other transformer from ADA to DIANA. No such predicate now exists. The kinds of questions that such a predicate should help to answer include the following:

1. Is a given abstract syntax tree (AST) correct for a given Ada program?

2. What should be the value of each semantic attribute in a DIANA structure?

3. When is sharing permitted in the AST?

4. May the same node appear in several sequences?

We believe that one way to meet these needs is by first specifying the transformation from ADA to AST and then defining a predicate, say π_t on ASTs and DIANA such that for an AST τ and a DIANA structure δ the predicate

$$\pi_t(\tau, \delta)$$

returns true if the DIANA structure δ is a correct representation of the AST τ. This dichotomy appears useful.

Translation of ADA source to abstract syntax tree (AST) is a two-step process:

- Translation of ADA source to parse tree (PT). The latter is a tree in which each node is labeled with the name of a non-terminal from ADA's BNF definition and has as many offspring as clauses appear in the relevant definition of that non-terminal. Given a non-ambiguous BNF for ADA, such a tree is uniquely defined. Although the BNF in ADA's LRM is ambiguous, it is not difficult to create a non-ambiguous version that preserves all essential structure.

- Translation of PT to AST. This step, though somewhat harder to specify than the previous one, is not conceptually difficult.

We believe it is possible to describe the PT to AST transformation by using

an attribute grammar to specify the AST as an attribute of the root of the PT. The specification of the AST to DIANA transformation (including specification of the semantic attributes) is a much harder problem and is still open. We are exploring methods of attacking these problems.

1.2. Structure of the Document

Abstractly, an instance of the DIANA form of an ADA program is an attributed tree. The tree's structure is basically that of the abstract syntax tree defined in the AFD. Attributes of the nodes of this tree encode the results of semantic analysis. Operations defined on the DIANA abstract data type (see Chapter 4) provide the predicates, selectors, and constructors required to manipulate this tree and its attributes. The structure of this document reflects the several facets of the DIANA definition.

- First we define precisely the *domain* of the DIANA data type. We do so by specifying the set of abstract trees, their attributes, and various assertions about them (which actually appear as comments). This definition is done in two steps:
  - In Section 1.4 we describe the notation, called IDL, for exhibiting DIANA's definition.
  - In Chapter 2, we use the notation to define the actual trees and attributes[8].
- Second, we provide a rationale for some of the more subtle design decisions— particularly with respect to the attributes of nodes in the abstract tree. This rationale appears in Chapter 3.
- Third, we define the *operations* on the DIANA abstract type. This definition appears in Chapter 4, and again is done in two steps. First, we describe generically the nature of these operations. Second, we show how these operations can be realized in conventional programming languages by showing how an interface can be derived from the DIANA definition *and* by showing the specification part (except for the private part) of an ADA package that specifies just such an interface. We also show here how the interface is altered when additional attributes or nodes are introduced[9].

---

[8] Note that a particular implementation may define an extended domain (additional attributes). What we define here is a *required* and *adequate* set.

[9] Note that an implementation may define additional operations. Again, we merely define a required and adequate set here.

- Fourth, in Chapter 5, we define a canonical way to represent DIANA structures external to a computer.

- Finally, in Chapter 6, we discuss implementation issues and illustrate some of the various options that are available.

There are also six appendices. *Appendix I provides* the definition of the predefined environment for ADA compilations. In Appendix II we define the Abstract Syntax Tree from the AFD as a derivative of the DIANA representation. Appendix III describes how the source of an ADA program can be regenerated from the DIANA representation and includes a discussion of the normalizations of reconstructed source programs imposed by DIANA.

Appendices IV, V, and VI provide three summaries of the DIANA definition. These summaries provide an invaluable cross reference into the main definitions and should be an important aid to the reader.

There is an extensive index that lists separately topics, DIANA attributes, and DIANA node names.

1.3. *Attribution Principles of DIANA*

This section describes the general principles used to decide on the details of DIANA. A more detailed rationale is given in Chapter 3.

The design of an intermediate representation involves deciding what information to represent explicitly and what information to recompute from the stored information. There are two extreme positions one can take:

- The source program (or its abstract syntax tree) contains all the necessary information; other information can be recomputed when necessary.

- All information which *can* be computed *should* be computed and stored within the intermediate representation.

DIANA's underlying principles, which are a compromise between these extrema, can be derived from DIANA's intended role in an ADA Program Support Environment (APSE) [3]. We envisage DIANA as created by an ADA Front End, used as input to that Front End for separate compilation purposes, used as input to the compiler's Back End, and used (produced or consumed) by a variety of other tools of the APSE.

For all these tools DIANA should contain information that is both *sufficient* and

appropriate. There are two questions relevant to deciding, about a given attribute, whether or not to include it in the DIANA:

- Does the information the attribute contains belong in the intermediate representation?

- Should the information be represented explicitly, or should it be recomputed from the stored information?

We have used two criteria in deciding of a given attribute whether or not to include it:

- DIANA should contain only such information as would be typically discovered via *static* (as opposed to *dynamic*) semantic analysis of the original program.

- If information can be easily recomputed, it should be omitted.

These two points are discussed at length in the following two subsections.

First, however, a point must be made. Although the original DIANA design used the metric of ease of computation in deciding what attributes to include, the concept has been considerably expanded in revisions of DIANA and of this report. As a result, attributes now exist in DIANA which, according to this criterion, ought not to be there. We have elected to let them remain for two reasons: They are not sufficiently wrong to require fixing, and their removal would likely unduly impact existing DIANA users. Note that these are the first two principles enunciated in Section 1.1.2 on page 11.

1.3.1. Static Semantic Information

We believe that it is appropriate for DIANA to include only that information that is determined from *static* semantic analysis, and that DIANA should exclude information whose determination requires *dynamic* semantic analysis.

This decision affects the evaluation of non-static expressions and evaluation of exceptions. For example, the attribute *sm_value* should not be used to hold the value of an expression that is not static, even if an implementation's semantic analyzer is capable of evaluating some such expressions. Similarly, exceptions are part of the execution (*i.e.*, dynamic) semantics of ADA and should not be represented in DIANA. Thus the attribute *sm_value* is no longer used to represent an exception to be raised, as it was in an earlier version of DIANA.

Of course, an implementation that does compute these additional values may record the information by defining additional attributes. However, any DIANA

consumer that relies on these attributes cannot be considered a correct DIANA 'user', as defined in Section 1.1.3 on page 12.

1.3.2. What is 'Easy to Recompute'?

Part of the criteria for including an attribute in DIANA is that it should be omitted if it is easy to recompute from the stored information. We feel it is important to avoid such redundant encodings if DIANA is to remain an usefully implementable internal representation. Of course this guideline requires that we define this phrase, and we suggest that an attribute is easily computed if

- it requires visits to no more than three to four nodes; or
- it can be computed in one pass through the DIANA tree, and all nodes with this attribute can be computed in the same pass.

The first criterion is clear; the second requires discussion.

Consider first an attribute that is needed by a compiler front end (FE) to do semantic analysis. As the FE does its work, it is free to create extra (non-DIANA) attributes for its purposes. Thus the desired attributes can be created by those who need them. To require them is an imposition on implementations which use algorithms that do not require these particular pointers. If we add every attribute that anyone requires, everyone will be overwhelmed.

Consider now an attribute needed by a back end (BE) to do code generation. As long as the attribute can be determined in a single pass, the routine that reads in the DIANA can readily add it as it reads in the DIANA. Again, some implementors may not need the attribute, and it is inappropriate to burden everyone with it.

It is for these reasons that we have rejected suggestions for pointers to the enclosing compilation unit, pointers to the enclosing namescope, and back pointers in general. These are attributes that are easily computed in one pass through the DIANA tree and indeed may not be needed by all implementations.

Of course, a DIANA producer can create a structure with extra attributes beyond those specified for DIANA. Nevertheless, any DIANA consumer that relies on these additional attributes is not a DIANA user, as that concept is defined in Section 1.1.3 on page 12.

1.3.3. Other Principles

There are other reasons why particular classes of attributes are present in DIANA.

- A tree-like representation of the source program is well-suited for many of the tools that will exist in an APSE, such as semantic analyzers, optimizers, and syntax-directed editors. The tree structure is represented in DIANA via the *structural* attributes; we use the same abstract syntax tree as given by the AFD, with a few differences described in Section 3.1 on page 80.

- *Lexical* attributes (such as symbol and literal representations and source positions) are needed by the compiler (*e.g.*, for error messages). They are also useful to other APSE tools for referring back to the source or for regenerating source text from the intermediate representation.

- ADA provides the attribute 'SIZE to determine the minimum number of bits needed to represent some object or subtype. If this attribute is applied to a static type, the result is static and is therefore required by ADA's semantics to be known at compile time. It represents a target-machine property properly computed by a code generator. However, as it can be used in static expressions, the Front End must know its value in some contexts. For example, the selection of a base type for a derived integer type depends on a range constraint. Without this information, the semantic analyzer cannot perform one of its most important tasks, type and overload resolution. Since the value must be known to the Front End, it is recorded as the value of an attribute to avoid the need for recomputation by the Back End.

1.3.4. Examples

A few examples illustrate these principles:

- The structure of a type (whether it is an integer, an array, a record, and so on) can be deduced by searching backward through the chain of derived types and subtypes. This chain could be of arbitrary length, and so the search is not tolerable. Thus, a subtype specification (a DIANA constrained node) has an attribute *sm_type_struct* to record this information.

- The parent type of a derived type is identical to the base type of the subtype indication given in the derived type definition, and this information is already recorded in the *sm_base_type* attribute of the constrained node which is a child of the derived node. Thus no parent type indication is needed in the derived node.

- Some DIANA users have suggested adding an attribute to each DEF_OCCURRENCE node to denote the node for the enclosing namescope. Although locating the enclosing namescope (if the at-

tribute is not available) can involve visits to *more than three* or four nodes, a DIANA reader can readily compute this attribute and decorate the tree with it as the DIANA is read in. Since this attribute contains information that may *not be useful* to *every* implementation and furthermore is easy to compute in the above sense, it is not provided in DIANA.

1.4. Notation

As we have stated several times, DIANA is an abstract data type that can be modeled as an attributed tree. In this document we are concerned with defining this abstract type—both its domain and its operations. The domain of the DIANA type is a subset of the (mathematical) domain known as attributed trees. In order to specify this subset precisely, we introduce some special notation, a subset of a notation called IDL [9]. A knowledge of IDL is not necessary to read or understand this document—all necessary information about the notation is defined in this section. (A few additional features are defined in Appendix II as they are used only there.)

To assist the reader in understanding this material, certain typographic conventions are followed consistently throughout this document, as illustrated in Figure 1-1.

DECL OP DEF_OCCURRENCE	IDL class names
constant var const_id	IDL *node names*
lx_srcpos sm_address as_exp	IDL attributes
Structure Root Type	reserved word in IDL
begin case pragma	ADA reserved words
INTEGER 'SIZE BOOLEAN	identifier defined by ADA
Tax_rate Walk1 Tree	identifier in an ADA program

Figure 1-1: Typographic Conventions used in this Document

The set of abstract trees used to model the DIANA type can be viewed as a language, one whose terminal sentences happen to be attributed trees rather than strings of characters. The definition of this language can, therefore, be given in a form similar to BNF. In particular, we use two definitional forms that

resemble the production rules of BNF. The first of these defines non-terminals of the description. Consider, for example, the following definition:

```
EXP ::= leaf | tree ;
```

As is customary, this definition may be read, 'The notion of an EXP is defined to be either a leaf or a tree.' Symbols such as EXP are called *class names*; names of nodes in the 'tree language' are called *node names*. In this case, both alternative definitions for EXP are *node names*. Class names, like non-terminals in BNF, never appear in the sentences of the language; their only use is in defining that language. Node names, on the other hand, appear in the sentences (that is the trees of our tree language). Notice, by the way, that each definitional rule is terminated by a semicolon.

The use of this form of definition is more restricted than in usual BNF. The right hand side of the production may be *only* an alternation of one or more class or *node names* and may not be a concatenation of two items (as it may be in BNF). In addition, class names may not depend upon themselves (in a circular fashion) involving only the '::=' form of definition rules. Thus a directed graph constructed with an edge from each class name on the left to each alternate on the right will be acyclic; that is, it will be a DAG.

As is usual with BNF, there may be more than one such production with the same left hand side (class name); definitions after the first merely introduce additional alternatives. Thus, the effect of the two definitions

```
EXP ::= leaf ;
...
EXP ::= tree ;
```

is no different from that of the single definition given earlier.

The definition of the node names must specify the attributes that are present in that node, as well as the names and types of these attributes. We again use a BNF-like form for such definitions. To prevent confusion, this form is slightly different from the definition of class names; for example

tree => *op*: OPERATOR, *left*: EXP, *right*: EXP ;

Here we define the node tree and associate with it three attributes with their names (*op*, *left*, and *right*) and their respective types (OPERATOR, EXP, and EXP). Unlike BNF (or record declarations), the order of attribute specifications does not matter.

The right hand side of a production defining a node name is also restricted: it may be *only* a sequence of zero or more attribute specifications separated by

commas and terminated by a semicolon. Multiple definitions of a node name are permitted: definitions after the first add additional attribute specifications—they are *not* alternatives but, rather, define additional attributes of the node. Thus, for example,

```
tree => op: OPERATOR ;
...
tree => left: EXP, right: EXP ;
```

and

```
tree => op: OPERATOR ;
...
tree => right: EXP ;
...
tree => left: EXP ;
```

are both identical in effect to the single definition given earlier. Note also that the order of both the `::=' and `=>' definition rules is irrelevant: all orders are equivalent (as in BNF). In particular, we reversed the order of definition of the *left* and *right* attributes in the last example above: doing so has no effect.

On occasion it is useful to specify a node which has no attributes, as for example

```
foo => ;
```

Nodes so defined are used much like enumeration literals in ADA. See, for example, the nodes **plus**, **minus**, **times**, and **divide** in Figure 1-2 on page 26.

There are two kinds of permissible attribute types: basic types defined by the IDL notation, and *private* types. The basic types are:

`Boolean`	These are the conventional boolean type: the only permissible values of such an attribute are `true` and `false`.
`Integer`	This is the `universal integer' type.
`Rational`	This is the `universal rational number' type, which includes all values typically found in computer integer, floating point and fixed point types.
`String`	These are ASCII strings.
`Seq Of` *T*	This is an ordered sequence of objects of type *T*.
<name>	The <name> *must* be that of either a node or class name defined elsewhere. Use of <name> as an attribute type denotes a reference to either an instance of that node (in the case of a node name) or any of the nodes that can be derived from it (in the case of a class name). Note that

a reference here does *not* necessarily mean a pointer in the concrete implementation; direct inline inclusion of the node is permitted, as well as a number of other implementations. (See Chapter 6 for a discussion of some of the implementation alternatives.)

A private type names an implementation-specific data structure that is inappropriate to specify at the abstract structure level. For example, in DIANA we want to associate a *source_position* attribute with each node of the abstract tree. This attribute is useful for reconstructing the source program, for reporting errors, for source-level debuggers, and so on. It is not a type, however, that should be defined as part of this standard since each computer system has idiosyncratic notions of how a position in the source program is encoded. For that matter, the concept of source position may not be meaningful if the DIANA arises from a syntax editor. For these reasons, attributes such as source position are merely defined to be private types.

A private type is introduced by a type declaration. The declaration of the private type 'MyType' would be

```
Type MyType;
```

Once such a declaration has been given, the type name may be used in an attribute specification. For example,

```
tree => xxx: MyType ;
```

Before proceeding, we need to make a few remarks about the lexical structure of the IDL notation. First, as in ADA a comment is introduced by a double hyphen '--' and is terminated by the end of the line on which it appears. Second, the notation is case sensitive; that is, identifiers that are spelled identically except for the case of the letters in them are considered to be different identifiers[10]. Finally, names (identifiers), as in ADA, consist of a letter followed by an optional sequence of letters, digits, and isolated underscore characters ('_').

The final point to be made about the notation is that the definitional rules illustrated above are enclosed in a syntactic structure that provides a name for the entity being defined together with the type of the goal symbol of the grammar. For example, the IDL text

---

[10] Case sensitivity is viewed by some as a questionable notational property; in this instance it was adopted to support a direct correspondence with the AFD (which is case sensitive).

```
Structure SomeName Root EXP Is
        -- some sequence of definitional rules
        End
```

asserts that the collection of production rules defines an abstract type (or *Structure* in IDL terminology) named 'SomeName' and that the root of this structure is an EXP, where EXP is defined by the set of definitional rules. In the case of DIANA we are defining a single abstract type, so there is a single occurrence of this syntax that surrounds the entire DIANA definition: other uses of IDL may require several Structure definitions. Expanding on the analogy that IDL is like BNF, the Root defined here is the 'goal symbol' of the grammar: all valid instances of the type defined by the IDL specification are derived by expanding this symbol.

1.4.1. Example of the IDL Notation

The following example illustrates the use of the notation. It is intentionally chosen *not* to be DIANA to avoid confusion. Suppose, then, that we wish to describe an abstract type for representing simple arithmetic expressions. We might use a definition such as the one shown in Figure 1-2 on page 26. Although this example is quite simple, it does illustrate the use of all of the features of the IDL notation that are *used to define* DIANA. Two class names are defined: EXP and OPERATOR. Since they name classes and not nodes (as indicated by our typographic conventions), neither appears in the trees in the abstract type (structure) 'ExpressionTree'. There are six node names defined: tree, leaf, plus, minus, times, and divide. Each of these may appear as a node in the trees. Of the nodes, only trees and leafs have attributes, and the names and types of these attributes are given. An implementation-defined private type, Source_Position, is defined; both trees and leafs have attributes of this type. Finally, the fact that the *root* of the tree must be an EXP, that is, either a tree or a leaf node, is specified. Figure 1-3 on page 27 illustrates several trees that are defined members of ExpressionTree; for expository reasons the names of the attributes and the source position attribute have been deleted from these pictures. Similar conventions are used in the figures in Chapter 3.

1.4.2. Specification of Representations

IDL can be used to define a *refinement* of a structure as well as an abstract data structure. A refinement is treated the same as any other abstract structure specified in IDL. A refinement of a structure is used to provide more detail about the abstract structure. In this document we define a refinement of DIANA

```
Structure ExpressionTree Root EXP Is

      -- First we define a private type.

      Type Source_Position;

      -- Next we define the notion of an expression, EXP.

      EXP  ::=  leaf | tree ;

      -- Next we define the nodes and their attributes.

      tree  =>  op: OPERATOR,  left: EXP,  right: EXP ;
      tree  =>  src: Source_Position ;
      leaf  =>  name: String ;
      leaf  =>  src: Source_Position ;

      -- Finally we define the  notion of an OPERATOR as the
      -- union of a collection of nodes; the null => productions
      -- are needed to define the node types since
      -- node type names are never implicitly defined.

      OPERATOR  ::=  plus | minus | times | divide ;

      plus => ;   minus => ;   times => ; divide => ;

End
```

Figure 1-2: Example of IDL Notation

that provides representation information for the private types defined in DIANA.

IDL can be used to define the package that contains the internal representation of a private type. and can specify the external representation of a private type. We add this information to the DIANA abstract type in the structure Diana_Concrete in Chapter 2 on page 77.

The internal representation of a private type is described by a definition of the form

```
For MyType Use MyPackage;
```

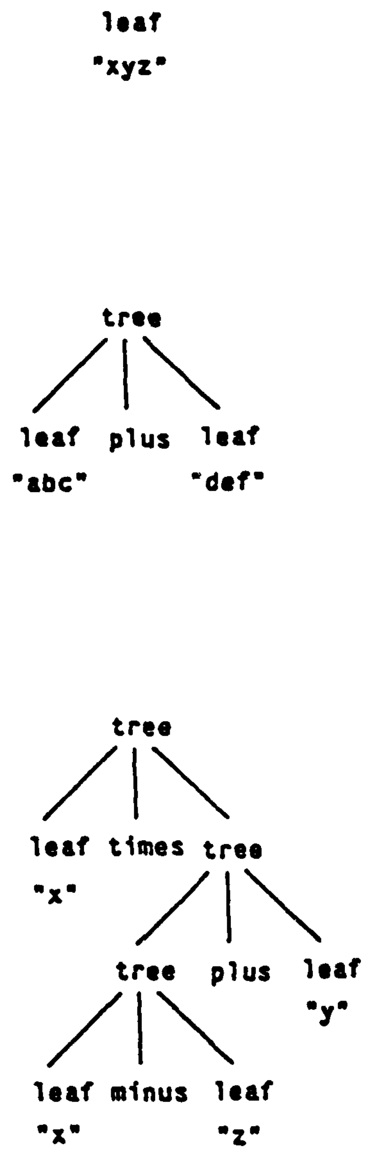


Figure 1-3: Some Trees in ExpressionTree

This definition introduces the name of the package (`MyPackage' in this case) where the definition of a private type (`MyType' in this case) is found.

The way a private type is to be represented externally can be described in a definition of the form

```
For MyType Use External ExternType;
```

This definition asserts that the private type MyType is represented by the type 'ExternType' externally. The external type may be one of the basic IDL types or a node type.

The refinement of a structure is specified with the following syntax

```
Structure AnotherTree Refines ExpressionTree Is
     -- Additional IDL statements to further define the
     -- structure ExpressionTree, such as a specification of the
     -- internal and external representations for private
     -- types in the abstract structure ExpressionTree.
     -- New nodes may be defined.

     End
```

1.4.3. Example of a Structure Refinement

The following example illustrates the use of the structure refinement notation. To continue with our example, suppose we wished to refine the abstract type ExpressionTree by adding an internal and external representation to be used for the private type Source_Position. We might refine the structure:

```
Structure AnotherTree Renames ExpressionTree Is

     -- first the internal representation of Source_Position

     For Source_Position Use Source_Package;

     -- next the external representation of Source_Position
     -- is given by a new node type, source_external_rep

     For Source_Position Use External source_external_rep;

     -- finally, we define the node type source_external_rep

     source_external_rep => file   : String,
                            line   : Integer,
                            char   : Integer;

End
```

This example completes the discussion of IDL. Notice that in the second example the internal representation and the external representation for the private type are both given. The internal representation is described in a separate package called Source_Package. The external representation is defined as a node, **source_external_rep**, that has three attributes, a file name, represented externally as a string, and a line number and character position, both of which are represented externally by the basic type 'Integer'. At the end of Chapter 2 we present a refinement Diana_Concrete of DIANA. In Chapter 5 we define the canonical external representation of DIANA.

1.4.4. DIANA Notational Conventions

The definition of DIANA given in the next chapter observes some notational conventions that are intended to improve the readability of the presentation. These include:

- Wherever reasonable, both nodes and classes are named as in the AFD.

- Typographic conventions are adhered to for class names, node names, and attributes to assist the reader. These conventions, which are are listed in Figure 1-1 on page 21, are that class names appear in all upper-case letters, nodes names in all lower case, and attribute names italicized.

- A class name or node name ending in '_S' or '_s' respectively is always a sequence of what comes before the '_'. Thus the reader can be sure on seeing a class name such as FOO_S that the definitions

```
FOO_S  =>   foo_s ;
foo_s  =>   as_list: Seq of FOO ;
```

appear somewhere.

- A class name ending in '_VOID' always has a definition such as

```
FOO_VOID => FOO | void ;
...
void      => ;
```

The node **void** has no attributes.

- There are four kinds of attributes defined in DIANA: *structural*, *lexical*, *semantic*, and *code*. The names of these attributes are lexically distinguished in the definition as follows:

*as_* structural attribute. The structural attributes define the abstract syntax tree of an ADA program. Their names are

those used in the AFD, prefixed with `*as_*'.

*lx_* lexical attributes. These provide information about the source form of the program, such as the spelling of identifiers or position in the source file.

*sm_* semantic attributes. These encode the results of semantic analysis—type and overload resolution, for example.

*cd_* code attributes—there is only one. This provides information from representation specifications that must be observed by the Back End.

• Although IDL is insensitive to the order of attribute definitions with `=>' rules, we have preserved the order used in the AFD. Additionally, for emphasis we have grouped structural, lexical, semantic, and code attributes, always in that order.

• The DIANA definition is organized in the same manner as the ADA LRM. To establish the correspondence, each set of DIANA rules begins with a comment that gives the corresponding section number of the ADA LRM and the concrete syntax defined there.

CHAPTER 2
DEFINITION OF THE DIANA DOMAIN

This chapter is devoted to the definition of the *domain of the* DIANA *abstract* type -- that is, to the definition of the set of attributed trees that model the values of the DIANA type. The definition is given in the notation discussed in section 1.4.

A simple refinement of the DIANA abstract structure follows the definition of the DIANA domain. This refinement defines the external representation of the private types used.

Before beginning the definition, which constitutes the bulk of this chapter, we make two observations about things that are *not* defined here.

- First, there are six private types used in the definition. Each of these corresponds to one of the kinds of information which may be installation or target machine specific. They include types for the source position of a node, the representation of identifiers, the representation of various values on the target system, and the representation of comments from the source program. The DIANA user must supply an implementation for each of these types.

- Second, as is explained in the ADA reference manual, a program is assumed to be compiled in a 'standard environment'. An ADA program may explicitly or implicitly reference entities defined in this environment, and the DIANA representation of the program must reflect this. The entities that may be referenced include the predefined attributes and types. The DIANA definition of these entities is not given here but is assumed to be available. See Appendix I for more details.

With these exceptions, the following completely defines the DIANA domain.

```
Structure Diana
Root COMPILATION is
--
--
--
--
--      Diana Reference Manual
--      ----- --------- ------
--
--
--    Version of 17 February 1983
--
--
--

Type source_position;
                          -- defines source position in original
                          -- source program;  used for error messages.
Type comments;
                          -- representation of comments;  used for
                          -- source reconstruction.
Type symbol_rep;
                          -- representation of identifiers,
                          -- strings, and characters
Type value;
                          -- implementation defined
                          -- gives value of a static expression;
                          -- can indicate that no value is computed.
Type operator;
                          -- enumeration type for all operators
                          -- used in implementation
Type number_rep;
                          -- representation of numeric literals
-- 2.  Lexical Elements
-- --------------------

-- Syntax 2.0
--     has no equivalent in concrete syntax
--


void =>                   :                    -- has no attributes

-- 2.3 Identifiers, 2.4 Numeric Literals, 2.6 String Literals

-- Syntax 2.3
--     not of interest for Diana
--

ID ::=                    DEF_ID | USED_ID;

OP ::=                    DEF_OP | USED_OP;

DESIGNATOR ::=            ID | OP;

DEF_OCCURRENCE ::=        DEF_ID | DEF_OP | DEF_CHAR;

-- see 4.4 for numeric_literal
```

```
-- 2.8  Pragmas
--     These productions do not correspond to productions in the
--     concrete syntax.

-- Syntax 2.8.A
-- pragma ::=
--     pragma identifier [(argument_association {, argument_association})];
--

    PRAGMA ::=             pragma;

    pragma =>              as_id               : ID,      -- a 'used_name_id'
                           as_param_assoc_s    : PARAM_ASSOC_S;
    pragma =>              lx_srcpos           : source_position,
                           lx_comments         : comments;

    PARAM_ASSOC_S ::=      param_assoc_s;

    param_assoc_s =>       as_list             : Seq Of PARAM_ASSOC;
    param_assoc_s =>       lx_srcpos           : source_position,
                           lx_comments         : comments;


-- Syntax 2.8.B
--  argument_association ::=
--       [argument_identifier =>] name
--     | [argument_identifier =>] expression
--

    -- see 6.4 for associations

-- 3.  Declarations and Types
--
-- 3.1  Declarations

-- Syntax 3.1
--  declaration ::=
--       object_declaration        | number_declaration
--     | type_declaration          | subtype_declaration
--     | subprogram_declaration    | package_declaration
--     | task_declaration          | generic_declaration
--     | exception_declaration     | generic_instantiation
--     | renaming_declaration      | deferred_constant_declaration
--

    DECL ::=               constant | var       -- object_declaration (3.2.A)
                           | number            -- number_declaration (3.2.B)
                           | type              -- type_declaration (3.3.1)
                           | subtype           -- subtype_declaration (3.3.2)
                           | subprogram_decl   -- subprogram_declaration (6.1)
                           | package_decl      -- package_declaration (7.1)
                           | task_decl         -- task_declaration (9.1)
                           | generic           -- generic_declaration (12.1)
                           | exception         -- exception_declaration (11.1)
                           -- See 12.3 for generic_instantiation,
                           -- See 8.5 for renaming_declaration,
                           | deferred_constant;        -- deferred_constant_declaration (7.4)

    DECL ::=               pragma;             -- pragma allowed as declaration
```

```
-- 3.2  Objects and Named Numbers

-- Syntax 3.2.A
--   object_declaration ::=
--       identifier_list : [constant] subtype_indication [:= expression];
--     | identifier_list : [constant] constrained_array_definition [:= expression];
--


    OBJECT_DEF ::=          EXP_VOID;
    EXP_VOID ::=            EXP | void;
    TYPE_SPEC ::=           CONSTRAINED;

    constant =>             as_id_s              : ID_S,   -- sequence of 'const_id'
                            as_type_spec         : TYPE_SPEC,
                            as_object_def        : OBJECT_DEF;
    constant =>             lx_srcpos            : source_position,
                            lx_comments          : comments;

    var =>                  as_id_s              : ID_S,   -- a sequence of 'var_id'
                            as_type_spec         : TYPE_SPEC,
                            as_object_def        : OBJECT_DEF;
    var =>                  lx_srcpos            : source_position,
                            lx_comments          : comments;


    DEF_ID ::=              var_id;

    var_id =>               lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    var_id =>               sm_obj_type          : TYPE_SPEC,
                            sm_address           : EXP_VOID,
                            sm_obj_def           : OBJECT_DEF;


    DEF_ID ::=              const_id;
      -- see  rationale Section 3.5.2 for discussion of deferred constants

    const_id =>             lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    const_id =>             sm_address           : EXP_VOID,
                            sm_obj_type          : TYPE_SPEC,
                            sm_obj_def           : OBJECT_DEF,
                            sm_first             : DEF_OCCURRENCE;      -- used for deferred
```

```
-- Syntax 3.2.B
--   number_declaration ::=
--       identifier_list : constant := universal_static_expression;
--


    number =>              as_id_s              : ID_S,   -- always a sequence of 'number_id'
                           as_exp               : EXP;
    number =>              lx_srcpos            : source_position,
                           lx_comments          : comments;


    DEF_ID ::=             number_id;

    number_id =>           lx_srcpos            : source_position,
                           lx_comments          : comments,
                           lx_symrep            : symbol_rep;
    number_id =>           sm_obj_type          : TYPE_SPEC,  -- always refers to a universal type
                           sm_init_exp          : EXP;


-- Syntax 3.2.C
--   identifier_list ::= identifier {, identifier}
--


    ID_S ::=               id_s;

    id_s =>                as_list              : Seq Of ID;
    id_s =>                lx_srcpos            : source_position,
                           lx_comments          : comments;


-- 3.3  Types and Subtypes
-- 3.3.1 Type Declarations


-- Syntax 3.3.1.A
--   type_declaration ::= full_type_declaration
--       | incomplete_type_declaration | private_type_declaration
--
--   full_type_declaration ::=
--       type identifier [discriminant_part] is type_definition;
--

    -- see 7.4 for private_type_declaration
    -- see 3.8.1 for incomplete_type_declaration

    type =>                as_id                : ID,      -- a 'type_id',
                                                           -- 'l_private_type_id' or
                                                           -- 'private_type_id'
                           as_dscrmt_var_s      : DSCRMT_VAR_S,  -- discriminant list, see 3.7.1
                           as_type_spec         : TYPE_SPEC;
    type =>                lx_srcpos            : source_position,
                           lx_comments          : comments;


    DEF_ID ::=             type_id;

    type_id =>             lx_srcpos            : source_position,
                           lx_comments          : comments,
                           lx_symrep            : symbol_rep;
    type_id =>             sm_type_spec         : TYPE_SPEC,
                           sm_first             : DEF_OCCURRENCE;     -- used for multiple def
```

```
-- Syntax 3.3.1.B
--   type_definition ::=
--       enumeration_type_definition   | integer_type_definition
--     | real_type_definition          | array_type_definition
--     | record_type_definition        | access_type_definition
--     | derived_type_definition
--

    TYPE_SPEC ::=          enum_literal_s       -- enumeration_type_definition (3.5.1)
                           | integer            -- integer_type_definition (3.5.4)
                           | fixed | float      -- real_type_definition (3.5.6)
                           | array              -- array_type_definition (3.6)
                           | record             -- record_type_definition (3.7)
                           | access             -- access_type_definition (3.8)
                           | derived;           -- derived_type_definition (3.4)
```

-- 3.3.2 Subtype Declarations

```
-- Syntax 3.3.2.A
--   subtype_declaration ::= subtype identifier is subtype_indication;
--

    subtype =>             as_id                : ID,
                           as_constrained       : CONSTRAINED;
    subtype =>             lx_srcpos            : source_position,
                           lx_comments          : comments;

    DEF_ID ::=             subtype_id;

    subtype_id =>          lx_srcpos            : source_position,
                           lx_comments          : comments,
                           lx_symrep            : symbol_rep;
    subtype_id =>          sm_type_spec         : CONSTRAINED;

-- Syntax 3.3.2.B

--   subtype_indication ::= type_mark [constraint]
--
--   type_mark ::= type_name | subtype_name
--

    CONSTRAINED ::=        constrained;
    CONSTRAINT ::=         void;

    constrained =>         as_name              : NAME,
                           as_constraint        : CONSTRAINT;
    constrained =>         lx_srcpos            : source_position,
                           lx_comments          : comments;
    constrained =>         sm_type_struct       : TYPE_SPEC,
                           sm_base_type         : TYPE_SPEC,
                           sm_constraint        : CONSTRAINT;
    constrained =>         cd_impl_size         : integer;
```

```
-- Syntax 3.3.2.C

--  constraint ::=
--        range_constraint | floating_point_constraint | fixed_point_constraint
--     | index_constraint | discriminant_constraint
--

    CONSTRAINT ::=        RANGE               -- range_constraint (3.5)
                          | float             -- floating_point_constraint (3.5.7)
                          | fixed             -- fixed_point_constraint (3.5.9)
                          | dscrt_range_s     -- index_constraint (3.6.C)
                          | dscrmt_aggregate;          -- discriminant_constraint (3.7.2)
```

-- 3.4 Derived Type Definitions

```
-- Syntax 3.4
--  derived_type_definition ::= new subtype_indication
--

    derived =>            as_constrained      : CONSTRAINED;
    derived =>            lx_srcpos           : source_position,
                          lx_comments         : comments;
    derived =>            sm_size             : EXP_VOID,
                          sm_actual_delta     : Rational,
                          sm_packing          : Boolean,
                          sm_controlled       : Boolean,
                          sm_storage_size     : EXP_VOID;
    derived =>            cd_impl_size        : Integer;
```

-- 3.5 Scalar Types

```
-- Syntax 3.5
--  range_constraint ::= range range
--
--  range ::= range_attribute
--     | simple_expression .. simple_expression
--

    RANGE ::=             range | attribute | attribute_call;

    range =>              as_exp1             : EXP,
                          as_exp2             : EXP;
    range =>              lx_srcpos           : source_position,
                          lx_comments         : comments;
    range =>              sm_base_type        : TYPE_SPEC;
```

-- 3.5.1 Enumeration Types

```
-- Syntax 3.5.1.A
--  enumeration_type_definition ::=
--     (enumeration_literal_specification {, enumeration_literal_specification})
--

    enum_literal_s =>     as_list             : Seq Of ENUM_LITERAL;
    enum_literal_s =>     lx_srcpos           : source_position,
                          lx_comments         : comments;
    enum_literal_s =>     sm_size             : EXP_VOID;
    enum_literal_s =>     cd_impl_size        : Integer;
```

```
-- Syntax 3.5.1.B
--   enumeration_literal_specification ::= enumeration_literal
--
--   enumeration_literal ::= identifier | character_literal
--


    ENUM_LITERAL ::=        enum_id | def_char;
    DEF_ID ::=              enum_id;
    DEF_CHAR ::=            def_char;

    enum_id =>              lx_srcpos         : source_position,
                            lx_comments       : comments,
                            lx_symrep         : symbol_rep;
    enum_id =>              sm_obj_type       : TYPE_SPEC,  -- refers to the 'enum_literal_s'
                            sm_pos            : integer,    -- consecutive position (base 0)
                            sm_rep            : integer;    -- user supplied representation value

    def_char =>             lx_srcpos         : source_position,
                            lx_comments       : comments,
                            lx_symrep         : symbol_rep;
    def_char =>             sm_obj_type       : TYPE_SPEC,  -- refers to the 'enum_literal_s'
                            sm_pos            : integer,    -- consecutive position (base 0)
                            sm_rep            : integer;    -- user supplied representation value
```

— 3.5.4 Integer Types

```
-- Syntax 3.5.4
--   integer_type_definition ::= range_constraint
--


    integer =>              as_range          : RANGE;
    integer =>              lx_srcpos         : source_position,
                            lx_comments       : comments;
    integer =>              sm_size           : EXP_VOID,
                            sm_type_struct    : TYPE_SPEC,
                            sm_base_type      : TYPE_SPEC; -- 'derived'
    integer =>              cd_impl_size      : integer;
```

— 3.5.6 Real Types

```
-- Syntax 3.5.6
--   real_type_definition ::=
--      floating_point_constraint | fixed_point_constraint
--

    -- see 3.5.7, 3.5.9
```

```
-- 3.5.7  Floating Point Types

-- Syntax 3.5.7
--   floating_point_constraint ::=
--      floating_accuracy_definition [range_constraint]
--
--   floating_accuracy_definition ::= digits static_simple_expression
--


    RANGE_VOID ::=        RANGE | void;

    float =>              as_exp             : EXP,
                          as_range_void      : RANGE_VOID;
    float =>              lx_srcpos          : source_position,
                          lx_comments        : comments;
    float =>              sm_size            : EXP_VOID,
                          sm_type_struct     : TYPE_SPEC,
                          sm_base_type       : TYPE_SPEC; -- 'derived'
    float =>              cd_impl_size       : integer;

-- 3.5.9  Fixed Point Types

-- Syntax 3.5.9
--   fixed_point_constraint  ::=
--      fixed_accuracy_definition [range_constraint]
--
--   fixed_accuracy_definition ::= delta static_simple_expression
--


    fixed =>              as_exp             : EXP,
                          as_range_void      : RANGE_VOID;
    fixed =>              lx_srcpos          : source_position,
                          lx_comments        : comments;
    fixed =>              sm_size            : EXP_VOID,
                          sm_actual_delta    : Rational,
                          sm_bits            : integer,
                          sm_base_type       : TYPE_SPEC; -- 'derived'
    fixed =>              cd_impl_size       : integer;
```

```
-- 3.6  Array Types

-- Syntax 3.6.A
--  array_type_definition ::=
--      unconstrained_array_definition | constrained_array_definition
--
--  unconstrained_array_definition ::=
--      array(index_subtype_definition {, index_subtype_definition}) of
--                   component_subtype_indication
--
--  constrained_array_definition ::=
--      array index_constraint of component_subtype_indication
--


    array =>            as_dscrt_range_s    : DSCRT_RANGE_S, -- index subtypes or constraint
                        as_constrained      : CONSTRAINED;   -- component subtype
    array =>            lx_srcpos           : source_position,
                        lx_comments         : comments;
    array =>            sm_size             : EXP_VOID,
                        sm_packing          : Boolean;

    DSCRT_RANGE_S ::=   dscrt_range_s;

    dscrt_range_s =>    as_list             : Seq Of DSCRT_RANGE;
    dscrt_range_s =>    lx_srcpos           : source_position,
                        lx_comments         : comments;

-- Syntax 3.6.B
--  index_subtype_definition ::= type_mark range <>
--


    DSCRT_RANGE ::=     index;

    index =>            as_name             : NAME;
    index =>            lx_srcpos           : source_position,
                        lx_comments         : comments;

-- Syntax 3.6.C
--  index_constraint ::= (discrete_range {, discrete_range})
--
--  discrete_range ::= discrete_subtype_indication | range
--


    DSCRT_RANGE ::=     constrained | RANGE;
```

-- 3.7 Record Types

```
-- Syntax 3.7.A
--   record_type_definition ::=
--      record
--          component_list
--      end record
--
```

REP_VOID ::=	REP | void;		
record =>	*as_list*	:	Seq Of COMP;
record =>	*lx_srcpos*	:	source_position,
	lx_comments	:	comments;
record =>	*sm_size*	:	EXP_VOID,
	sm_discriminants	:	DSCRMT_VAR_S,
	sm_packing	:	Boolean,
	sm_record_spec	:	REP_VOID;

```
-- Syntax 3.7.B
--   component_list ::=
--        component_declaration {component_declaration}
--      | {component_declaration} variant_part
--      | null;
--
--   component_declaration ::=
--      identifier_list : component_subtype_definition [:= expression];
--
--   component_subtype_definition ::= subtype_indication
--
```

COMP ::=	var	-- component_declaration (3.2) where ID is 'comp_id'
	| variant_part	-- variant_part (3.7.3.A)
	| *null_comp*;	-- null (see below)
COMP ::=	pragma;	-- pragmas are allowed in component declarations

null_comp =>	*lx_srcpos*	:	source_position,
	lx_comments	:	comments;
DEF_ID ::=	comp_id;		
COMP_REP_VOID ::=	COMP_REP | void;		
comp_id =>	*lx_srcpos*	:	source_position,
	lx_comments	:	comments,
	lx_symrep	:	symbol_rep;
comp_id =>	*sm_obj_type*	:	TYPE_SPEC,
	sm_init_exp	:	EXP_VOID,
	sm_comp_spec	:	COMP_REP_VOID;

```
-- 3.7.1  Discriminants

-- Syntax 3.7.1
--   discriminant_part ::=
--      (discriminant_specification {; discriminant_specification})
--
--   discriminant_specification ::=
--      identifier_list : type_mark [:= expression]
--


    DSCRMT_VAR_S ::=        dscrmt_var_s;

    dscrmt_var_s =>         as_list              : Seq Of DSCRMT_VAR;
    dscrmt_var_s =>         lx_srcpos            : source_position,
                            lx_comments          : comments;

    DSCRMT_VAR ::=          dscrmt_var;          -- where 'ID' is always a 'dscrmt_id'

    dscrmt_var =>           as_id_s              : ID_S,  -- a sequence of 'var_id'
                            as_name              : NAME,
                            as_object_def        : OBJECT_DEF;
    dscrmt_var =>           lx_srcpos            : source_position,
                            lx_comments          : comments;

    DEF_ID ::=              dscrmt_id;

    dscrmt_id =>            lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    dscrmt_id =>            sm_obj_type          : TYPE_SPEC,
                            sm_init_exp          : EXP_VOID,
                            sm_first             : DEF_OCCURRENCE,
                            sm_comp_spec         : COMP_REP_VOID;

-- 3.7.2  Discriminant Constraints

-- Syntax 3.7.2
--   discriminant_constraint ::=
--      (discriminant_association {, discriminant_association})
--   discriminant_association ::=
--      [discriminant_simple_name {| discriminant_simple_name} =>] expression
--


    dscrmt_aggregate =>     as_list              : Seq Of COMP_ASSOC;
    dscrmt_aggregate =>     lx_srcpos            : source_position,
                            lx_comments          : comments;
    dscrmt_aggregate =>     sm_normalized_comp_s         : EXP_S;

    -- see 4.3.B for discriminant association
```

```
-- 3.7.3 Variant Parts

-- Syntax 3.7.3.A
--   variant_part ::=
--       case discriminant_simple_name is
--           variant
--           {variant}
--       end case;
--
--   variant  ::=
--       when choice {| choice} =>
--           component_list
--


    variant_part =>          as_name          : NAME,
                             as_variant_s     : VARIANT_S;
    variant_part =>          lx_srcpos        : source_position,
                             lx_comments      : comments;


    VARIANT_S ::=            variant_s;

    variant_s =>             as_list          : Seq Of VARIANT;
    variant_s =>             lx_srcpos        : source_position,
                             lx_comments      : comments;

    VARIANT ::=              variant;
    CHOICE_S ::=             choice_s;
    INNER_RECORD ::=         inner_record;

    choice_s =>              as_list          : Seq Of CHOICE;
    choice_s =>              lx_srcpos        : source_position,
                             lx_comments      : comments;

    variant =>               as_choice_s      : CHOICE_S,
                             as_record        : INNER_RECORD;
    variant =>               lx_srcpos        : source_position,
                             lx_comments      : comments;

    inner_record =>          as_list          : Seq Of COMP;
    inner_record =>          lx_srcpos        : source_position,
                             lx_comments      : comments;


-- Syntax 3.7.3.B
--   choice ::= simple_expression
--       | discrete_range | others | component_simple_name
--


    CHOICE ::=               EXP | DSCRT_RANGE | others;

    others =>                lx_srcpos        : source_position,
                             lx_comments      : comments;
```

```
-- 3.8  Access Types

-- Syntax 3.8
--  access_type_definition ::= access subtype_indication
--

    access =>               as_constrained       : CONSTRAINED;
    access =>               lx_srcpos            : source_position,
                            lx_comments          : comments;
    access =>               sm_size              : EXP_VOID,
                            sm_storage_size      : EXP_VOID,
                            sm_controlled        : Boolean;

    -- See 4.4.C for null access value

-- 3.8.1 Incomplete Type Declarations

-- Syntax 3.8.1
--  incomplete_type_declaration ::= type identifier [discriminant_part];
--

    TYPE_SPEC ::=           void;

    --  incomplete types are described in the rationale Section 3.5.1.1

-- 3.9  Declarative Parts

-- Syntax 3.9.A
--  declarative_part ::=
--      {basic_declarative_item} {later_declarative_item}
--
--  basic_declarative_item ::= basic_declaration
--      | representation_clause | use_clause
--

    DECL ::=                REP | use;      --representation is declarative item

-- Syntax 3.9.B
--  later_declarative_item ::= body
--      | subprogram_declaration  | package_declaration
--      | task_declaration        | generic_declaration
--      | use_clause              | generic_instantiation
--
--  body  ::=  proper_body | stub
--
--  proper_body ::= subprogram_body | package_body | task_body
--

    ITEM_S ::=              item_s;
    ITEM ::=                DECL | subprogram_body | package_body | task_body;
    -- see 3.1, 6.1, 7.1, 9.1, 10.2 (stub included in _body definitions)

    item_s =>               as_list              : Seq Of ITEM;
    item_s =>               lx_srcpos            : source_position,
                            lx_comments          : comments;
```

```
-- 4.  Names and Expressions
--
-- 4.1  Names

-- Syntax 4.1.A
--   name ::= simple_name
--       | character_literal     | operator_symbol
--       | indexed_component     | slice
--       | selected_component    | attribute
--
--   simple_name ::= identifier
--

    NAME ::=             DESIGNATOR            -- identifier and operator (2.3)
                         | used_char           -- character_literal (see below)
                         | indexed             -- indexed_component (4.1.1)
                         | slice               -- slice (4.1.2)
                         | selected | all      -- selected_component (4.1.3)
                         | attribute | attribute_call;  -- attribute (4.1.4)

    USED_ID ::=          used_object_id | used_name_id | used_bltn_id;

    used_object_id =>    lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_object_id =>    sm_exp_type          : TYPE_SPEC,
                         sm_defn              : DEF_OCCURRENCE,
                         sm_value             : value;

    used_name_id =>      lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_name_id =>      sm_defn              : DEF_OCCURRENCE;

    used_bltn_id =>      lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_bltn_id =>      sm_operator          : operator;

    -- see 3.8.5 of rationale for a discussion of built-in subprograms

    USED_OP ::=          used_op | used_bltn_op;

    used_op =>           lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_op =>           sm_defn              : DEF_OCCURRENCE;

    used_bltn_op =>      lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_bltn_op =>      sm_operator          : operator;

    used_char =>         lx_srcpos            : source_position,
                         lx_comments          : comments,
                         lx_symrep            : symbol_rep;
    used_char =>         sm_defn              : DEF_OCCURRENCE,
                         sm_exp_type          : TYPE_SPEC,
                         sm_value             : value;

-- Syntax 4.1.B
--   prefix ::= name | function_call
--

    NAME ::=             function_call;        -- see 6.4
```

— 4.1.1 Indexed Components

```
-- Syntax 4.1.1
--   indexed_component ::= prefix(expression {, expression})
--

    EXP_S ::=            exp_s;

    exp_s =>             as_list            : Seq Of EXP;
    exp_s =>             lx_srcpos          : source_position,
                         lx_comments        : comments;

    indexed =>           as_name            : NAME,
                         as_exp_s           : EXP_S;
    indexed =>           lx_srcpos          : source_position,
                         lx_comments        : comments;
    indexed =>           sm_exp_type        : TYPE_SPEC;
```

— 4.1.2 Slices

```
-- Syntax 4.1.2
--   slice ::= prefix(discrete_range)
--

    slice =>             as_name            : NAME,
                         as_dscrt_range     : DSCRT_RANGE;
    slice =>             lx_srcpos          : source_position,
                         lx_comments        : comments;
    slice =>             sm_exp_type        : TYPE_SPEC,
                         sm_constraint      : CONSTRAINT;
```

— 4.1.3 Selected Components

```
-- Syntax 4.1.3
--   selected_component ::=  prefix.selector
--
--   selector ::= simple_name
--       | character_literal | operator_symbol | all
--

    DESIGNATOR_CHAR::=   DESIGNATOR | used_char;    -- character literals allowed as selector

    selected =>          as_name            : NAME,
                         as_designator_char : DESIGNATOR_CHAR;
    selected =>          lx_srcpos          : source_position,
                         lx_comments        : comments;
    selected =>          sm_exp_type        : TYPE_SPEC;

    all =>               as_name            : NAME; -- used for name.all
    all =>               lx_srcpos          : source_position,
                         lx_comments        : comments;
    all =>               sm_exp_type        : TYPE_SPEC;
```

```
-- 4.1.4 Attributes

-- Syntax 4.1.4
--   attribute ::= prefix'attribute_designator
--
--   attribute_designator ::= simple_name [(universal_static_expression)]
--

    attribute =>          as_name            : NAME,
                          as_id              : ID;    -- always a 'used_name_id',
                                                      -- whose attributes point to
                                                      -- a predefined 'attr_id'
    attribute =>          lx_srcpos          : source_position,
                          lx_comments        : comments;
    attribute =>          sm_exp_type        : TYPE_SPEC,
                          sm_value           : value;

    attribute_call =>     as_name            : NAME,  -- used for attributes
                                                      -- with arguments
                                                      -- NAME can only be attribute
                          as_exp             : EXP;
    attribute_call =>     lx_srcpos          : source_position,
                          lx_comments        : comments;
    attribute_call =>     sm_exp_type        : TYPE_SPEC,
                          sm_value           : value;

-- 4.2 Literals

-- Refer to 4.4.C for numeric_literal, string_literal,
-- and null_access.
-- Refer to 4.1 for character_literal

    -- The enumeration_literal is represented as a 'used_object_id' or a
    -- 'used_char' whose attributes point to an 'enum_id' or a 'def_char'.
    -- See 3.5.1.B


-- 4.3 Aggregates

-- Syntax 4.3.A
--   aggregate ::=
--      (component_association {, component_association})
--

    EXP ::=               aggregate;

    aggregate =>          as_list            : Seq Of COMP_ASSOC;
    aggregate =>          lx_srcpos          : source_position,
                          lx_comments        : comments;
    aggregate =>          sm_exp_type        : TYPE_SPEC,
                          sm_constraint      : CONSTRAINT,
                          sm_normalized_comp_s     : EXP_S;

-- Syntax 4.3.B
--   component_association ::=
--      [choice {| choice} => ] expression
--

    COMP_ASSOC ::=        named | EXP;

    named =>              as_choice_s        : CHOICE_S,
                          as_exp             : EXP;
    named =>              lx_srcpos          : source_position,
                          lx_comments        : comments;
```

— 4.4 Expressions

```
— Syntax 4.4.A
—  expression ::=
—        relation {and relation}   | relation {and then relation}
—      | relation {or relation}    | relation {or else relation}
—      | relation {xor relation}
—


EXP ::=                 binary;            — only for short-circuit
                                           — expressions; see 3.3.4 of rationale

binary =>               as_exp1            : EXP,
                        as_binary_op       : BINARY_OP,
                        as_exp2            : EXP;
binary =>               lx_srcpos          : source_position,
                        lx_comments        : comments;
binary =>               sm_exp_type        : TYPE_SPEC, — always the TYPE_SPEC
                                           — of a Boolean type
                        sm_value           : value;

BINARY_OP ::=           SHORT_CIRCUIT_OP;
SHORT_CIRCUIT_OP ::=    and_then | or_else;

and_then =>             lx_srcpos          : source_position,
                        lx_comments        : comments;
or_else  =>             lx_srcpos          : source_position,
                        lx_comments        : comments;


— Syntax 4.4.B
—  relation ::=
—        simple_expression [relational_operator simple_expression]
—      | simple_expression [not] in range
—      | simple_expression [not] in type_mark
—


EXP ::=                 membership;
TYPE_RANGE ::=          RANGE | NAME;

membership =>           as_exp             : EXP,
                        as_membership_op   : MEMBERSHIP_OP,
                        as_type_range      : TYPE_RANGE;
membership =>           lx_srcpos          : source_position,
                        lx_comments        : comments;
membership =>           sm_exp_type        : TYPE_SPEC, — always the TYPE_SPEC
                                           — of a Boolean type
                        sm_value           : value;

MEMBERSHIP_OP ::=       in_op | not_in;

in_op =>                lx_srcpos          : source_position,
                        lx_comments        : comments;
not_in =>               lx_srcpos          : source_position,
                        lx_comments        : comments;
```

```
-- Syntax 4.4.C
--  simple_expression ::=
--     [unary_operator] term {binary_adding_operator term}
--
--  term ::= factor {multiplying_operator factor}
--
--  factor ::= primary [** primary] | abs primary | not primary
--


-- Syntax 4.4.D
--  primary ::=
--       numeric_literal | null | aggregate | string_literal | name | allocator
--     | function_call | type_conversion | qualified_expression | (expression)
--

    EXP ::=               NAME                 -- name, function_call (4.1, 6.4)
                          | numeric_literal    -- numeric_literal (below)
                          | null_access        -- null (see below)
                          | aggregate          -- aggregate (4.3)
                          | string_literal     -- string_literal (below)
                          | allocator          -- allocator (4.8)
                          | conversion         -- type_conversion (4.6)
                          | qualified          -- qualified_expression (4.7)
                          | parenthesized;     -- (expression) (below)

    -- This is not a construct in the Formal Definition.
    -- See rationale

    parenthesized =>      as_exp               : EXP;
    parenthesized =>      lx_srcpos            : source_position,
                          lx_comments          : comments;
    parenthesized =>      sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;

    numeric_literal =>    lx_srcpos            : source_position,
                          lx_comments          : comments,
                          lx_numrep            : number_rep;
    numeric_literal =>    sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;
    -- if there is implicit conversion sm_exp_type reflects conversion;
    -- otherwise it references a universal type

    string_literal =>     lx_srcpos            : source_position,
                          lx_comments          : comments,
                          lx_symrep            : symbol_rep;
    string_literal =>     sm_exp_type          : TYPE_SPEC,
                          sm_constraint        : CONSTRAINT,
                          sm_value             : value;

    null_access =>        lx_srcpos            : source_position,
                          lx_comments          : comments;
    null_access =>        sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;
```

— 4.5 Operators and Expression Evaluation

```
— Syntax 4.5
—   logical_operator ::= and | or | xor
—
—   relational_operator ::=  = | /= | < | <= | > | >=
—
—   adding_operator ::= + | - | &
—
—   unary_operator ::= + | -
—
—   multiplying_operator ::= * | / | mod | rem
—
—   highest_precedence_operator ::= ** | abs | not
—

    — operators are incorporated in function calls, see 3.3.4 of rationale
    — operators are defined in Diana refinement, Diana_Concrete
```

— 4.6 Type Conversions

```
— Syntax 4.6
—   type_conversion ::= type_mark(expression)
—

    conversion =>         as_name              : NAME,
                          as_exp               : EXP;
    conversion =>         lx_srcpos            : source_position,
                          lx_comments          : comments;
    conversion =>         sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;
```

— 4.7 Qualified Expressions

```
— Syntax 4.7
—   qualified_expression ::=
—       type_mark'(expression) | type_mark'aggregate
—

    qualified =>          as_name              : NAME,
                          as_exp               : EXP;
    qualified =>          lx_srcpos            : source_position,
                          lx_comments          : comments;
    qualified =>          sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;
```

— 4.8 Allocators

```
— Syntax 4.8
—   allocator ::=
—       new subtype_indication | new qualified_expression
—

    EXP_CONSTRAINED::=    EXP | CONSTRAINED;

    allocator =>          as_exp_constrained   : EXP_CONSTRAINED;
    allocator =>          lx_srcpos            : source_position,
                          lx_comments          : comments;
    allocator =>          sm_exp_type          : TYPE_SPEC,
                          sm_value             : value;
```

```
-- 5.  Statements
--
-- 5.1  Simple and Compound Statements - Sequences of Statements

-- Syntax 5.1.A
--   sequence_of_statements ::= statement {statement}
--

    STM_S ::=              stm_s;

    stm_s =>               as_list             : Seq Of STM;
    stm_s =>               lx_srcpos           : source_position,
                           lx_comments         : comments;


-- Syntax 5.1.B
--   statement ::=
--      {label} simple_statement | {label} compound_statement
--

    STM ::=                labeled;

    labeled =>             as_id_s             : ID_S,   -- Seq of 'label_id'
                           as_stm              : STM;
    labeled =>             lx_srcpos           : source_position,
                           lx_comments         : comments;

    DEF_ID ::=             label_id;

    label_id =>            lx_srcpos           : source_position,
                           lx_comments         : comments,
                           lx_symrep           : symbol_rep;
    label_id =>            sm_stm              : STM;    -- always 'labeled'


-- Syntax 5.1.C
--   simple_statement ::= null_statement
--        | assignment_statement | procedure_call_statement
--        | exit_statement       | return_statement
--        | goto_statement       | entry_call_statement
--        | delay_statement      | abort_statement
--        | raise_statement      | code_statement
--

    STM ::=                null_stm            -- null_statement (5.1.F)
                           | assign            -- assignment_statement (5.2)
                           | procedure_call    -- procedure_call_statement (6.4)
                           | exit              -- exit_statement (5.7)
                           | return            -- return_statement (5.8)
                           | goto              -- goto_statement (5.9)
                           | entry_call        -- entry_call_statement (9.5.B)
                           | delay             -- delay_statement (9.6)
                           | abort             -- abort_statement (9.10)
                           | raise             -- raise_statmment (11.3)
                           | code;             -- code_statement (13.8)

    STM ::=                pragma;             -- pragma allowed where
                                               -- statement allowed
```

```
-- Syntax 5.1.D
--   compound_statement ::=
--        if_statement          | case_statement
--      | loop_statement        | block_statement
--      | accept_statement      | select_statement
--

    STM ::=               if                            -- if_statment (5.3)
                        | case                          -- case_statement (5.4)
                        | named_stm | LOOP              -- loop_statement (5.5)
                        | block                         -- block_statement (5.6)
                        | accept                        -- accept_statement (9.5.C)
                        | select | cond_entry | timed_entry;
                                                        -- select_statement (9.7)


-- Syntax 5.1.E
--   label ::= <<label_simple_name>>
--

    -- see 5.1.B

-- Syntax 5.1.F
--   null_statement ::= null ;
--

    null_stm =>           lx_srcpos         : source_position,
                          lx_comments       : comments;
```

-- 5.2 Assignment Statement

```
-- Syntax 5.2
--   assignment_statement ::=
--      variable_name := expression;
--

    assign =>             as_name           : NAME,
                          as_exp            : EXP;
    assign =>             lx_srcpos         : source_position,
                          lx_comments       : comments;
```

— 5.3 If Statements

```
-- Syntax 5.3.A
--  if_statement ::=
--      if condition then
--          sequence_of_statements
--     {elsif condition then
--          sequence_of_statements}
--     [else
--          sequence_of_statements]
--      end if;
--


    if =>                   as_list              : Seq Of COND_CLAUSE;
    if =>                   lx_srcpos            : source_position,
                            lx_comments          : comments;

    COND_CLAUSE ::=         cond_clause;

    cond_clause =>          as_exp_void          : EXP_VOID,    --void for else
                            as_stm_s             : STM_S;
    cond_clause =>          lx_srcpos            : source_position,
                            lx_comments          : comments;

-- Syntax 5.3.B
--  condition ::= boolean_expression
--

    -- condition is replaced by EXP
```

— 5.4 Case Statements

```
-- Syntax 5.4
--  case_statement ::=
--     case expression is
--          case_statement_alternative
--         {case_statement_alternative}
--     end case;
--
--  case_statement_alternative ::=
--     when choice {| choice } =>
--          sequence_of_statements)
--


    ALTERNATIVE_S ::=       alternative_s;
    ALTERNATIVE ::=         alternative | pragma;        -- pragma allowed where alternative allowed

    case =>                 as_exp               : EXP,
                            as_alternative_s     : ALTERNATIVE_S;
    case =>                 lx_srcpos            : source_position,
                            lx_comments          : comments;

    alternative_s =>        as_list              : Seq Of ALTERNATIVE;
    alternative_s =>        lx_srcpos            : source_position,
                            lx_comments          : comments;

    alternative =>          as_choice_s          : CHOICE_S,
                            as_stm_s             : STM_S;
    alternative =>          lx_srcpos            : source_position,
                            lx_comments          : comments;
```

-- 5.5 Loop Statements

```
-- Syntax 5.5.A
--  loop_statement ::=
--     [loop_simple_name:]
--          [iteration_scheme] loop
--               sequence_of_statements
--          end loop [loop_simple_name];
--


    named_stm =>          as_id               : ID,      -- always a 'named_stm_id'
                          as_stm              : STM;     -- 'loop' or 'block'
    named_stm =>          lx_srcpos           : source_position,
                          lx_comments         : comments;

    DEF_ID ::=            named_stm_id;

    named_stm_id =>       lx_srcpos           : source_position,
                          lx_comments         : comments,
                          lx_symrep           : symbol_rep;
    named_stm_id =>       sm_stm              : STM;     -- always 'named_stm'


    LOOP ::=              loop;
    ITERATION ::=         void;

    loop =>               as_iteration        : ITERATION,
                          as_stm_s            : STM_S;
    loop =>               lx_srcpos           : source_position,
                          lx_comments         : comments;
```

```
-- Syntax 5.5.B
--  iteration_scheme ::=  while condition
--      | for loop_parameter_specification
--
--  loop_parameter_specification ::=
--      identifier in [reverse] discrete_range
--
```

```
ITERATION ::=           for | reverse;

for =>                  as_id            : ID,      -- always an 'iteration_id'
                        as_dscrt_range   : DSCRT_RANGE;
for =>                  lx_srcpos        : source_position,
                        lx_comments      : comments;

reverse =>              as_id            : ID,      -- always an 'iteration_id'
                        as_dscrt_range   : DSCRT_RANGE;
reverse =>              lx_srcpos        : source_position,
                        lx_comments      : comments;


DEF_ID ::=              iteration_id;

iteration_id =>         lx_srcpos        : source_position,
                        lx_comments      : comments,
                        lx_symrep        : symbol_rep;
iteration_id =>         sm_obj_type      : TYPE_SPEC;


ITERATION ::=           while;

while =>                as_exp           : EXP;
while =>                lx_srcpos        : source_position,
                        lx_comments      : comments;
```

-- 5.6 Block Statements

```
-- Syntax 5.6
--  block_statement ::=
--      [block_simple_name:]
--          [declare
--              declarative_part]
--          begin
--              sequence_of_statements
--          [exception
--              exception_handler
--              {exception_handler}]
--          end [block_simple_name];
--
```

```
-- see 5.5.A for named block

block =>                as_item_s        : ITEM_S,
                        as_stm_s         : STM_S,
                        as_alternative_s : ALTERNATIVE_S;
block =>                lx_srcpos        : source_position,
                        lx_comments      : comments;
```

```
-- 5.7  Exit Statements

-- Syntax 5.7
--   exit_statement ::=
--       exit [loop_name] [when condition];
--

    NAME_VOID ::=         NAME | void;

    exit =>               as_name_void      : NAME_VOID,
                          as_exp_void       : EXP_VOID;
    exit =>               lx_srcpos         : source_position,
                          lx_comments       : comments;
    exit =>               sm_stm            : LOOP;  -- Computed even when there
                                                     -- is no name given
                                                     -- in the source program.

-- 5.8  Return Statements

-- Syntax 5.8
--   return_statement ::= return [expression];
--

    return =>             as_exp_void       : EXP_VOID;
    return =>             lx_srcpos         : source_position,
                          lx_comments       : comments;

-- 5.9  Goto Statements

-- Syntax 5.9
--   goto_statement ::= goto label_name;
--

    goto =>               as_name           : NAME;
    goto =>               lx_srcpos         : source_position,
                          lx_comments       : comments;
```

```
-- 6.  Subprograms
--
-- 6.1  Subprogram Declarations

-- Syntax 6.1.A
--   subprogram_declaration ::= subprogram_specification;
--

    SUBPROGRAM_DEF ::=  void;

    -- for procedure and function subprogram designator is one of 'proc_id',
    -- 'function_id', or 'def_op'
    -- for entry subprogram designator is 'entry_id'
    -- for renaming can be any of above or 'enum_id' see 3.7 in rationale

    subprogram_decl =>      as_designator        : DESIGNATOR,
                            as_header            : HEADER,
                            as_subprogram_def    : SUBPROGRAM_DEF;
    subprogram_decl =>      lx_srcpos            : source_position,
                            lx_comments          : comments;

    DEF_ID ::=              proc_id;

    proc_id =>              lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    proc_id =>              sm_spec              : HEADER,
                            sm_body              : SUBP_BODY_DESC,
                            sm_location          : LOCATION,
                            sm_stub              : DEF_OCCURRENCE,
                            sm_first             : DEF_OCCURRENCE;

    DEF_ID ::=              function_id;

    function_id =>          lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    function_id =>          sm_spec              : HEADER,
                            sm_body              : SUBP_BODY_DESC,
                            sm_location          : LOCATION,
                            sm_stub              : DEF_OCCURRENCE,
                            sm_first             : DEF_OCCURRENCE;

    DEF_OP ::=              def_op;

    def_op =>               lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    def_op =>               sm_spec              : HEADER,
                            sm_body              : SUBP_BODY_DESC,
                            sm_location          : LOCATION,
                            sm_stub              : DEF_OCCURRENCE,
                            sm_first             : DEF_OCCURRENCE;

    LANGUAGE ::=            argument_id;
    LOCATION ::=            EXP_VOID | pragma_id;
    SUBP_BODY_DESC ::=      block | stub | instantiation |
                            FORMAL_SUBPROG_DEF | rename | LANGUAGE | void;

-- 'pragma_id' and 'argument_id' only occur in the predefined environment
```

```
-- Syntax 6.1.B
--   subprogram_specification ::=
--        procedure identifier [formal_part]
--      | function designator [formal_part] return type_mark
--
--   designator ::= identifier | operator_symbol
--
--   operator_symbol ::= string_literal
--
```

```
HEADER ::=           procedure;
HEADER ::=           function;

procedure =>         as_param_s       : PARAM_S;
procedure =>         lx_srcpos        : source_position,
                     lx_comments      : comments;

function =>          as_param_s       : PARAM_S,
                     as_name_void     : NAME_VOID;
                                      -- void in case of instantiation
function =>          lx_srcpos        : source_position,
                     lx_comments      : comments;
```

```
-- Syntax 6.1.C
-- formal_part ::=
--    (parameter_specification {; parameter_specification})
--
-- parameter_specification ::=
--    identifier_list : mode type_mark [:= expression]
--
-- mode  ::= [in] | in out | out
--


PARAM_S ::=          param_s;

param_s =>           as_list          : Seq Of PARAM;
param_s =>           lx_srcpos        : source_position,
                     lx_comments      : comments;

PARAM ::=            in;

in =>                as_id_s          : ID_S,  -- always a sequence of 'in_id'
                     as_name          : NAME,
                     as_exp_void      : EXP_VOID;
in =>                lx_srcpos        : source_position,
                     lx_comments      : comments,
                     lx_default       : Boolean;

PARAM ::=            in_out;
PARAM ::=            out;

in_out =>            as_id_s          : ID_S,  -- always a sequence of 'in_out_id'
                     as_name          : NAME,
                     as_exp_void      : EXP_VOID;  -- always void
in_out =>            lx_srcpos        : source_position,
                     lx_comments      : comments;

out =>               as_id_s          : ID_S,  -- always a sequence of 'out_id'
                     as_name          : NAME,
                     as_exp_void      : EXP_VOID;  -- always void
out =>               lx_srcpos        : source_position,
                     lx_comments      : comments;


DEF_ID ::=           in_id;

in_id =>             lx_srcpos        : source_position,
                     lx_comments      : comments,
                     lx_symrep        : symbol_rep;
in_id =>             sm_obj_type      : TYPE_SPEC,
                     sm_init_exp      : EXP_VOID,
                     sm_first         : DEF_OCCURRENCE;

DEF_ID ::=           in_out_id  |  out_id;

in_out_id =>         lx_srcpos        : source_position,
                     lx_comments      : comments,
                     lx_symrep        : symbol_rep;
in_out_id =>         sm_obj_type      : TYPE_SPEC,
                     sm_first         : DEF_OCCURRENCE;

out_id =>            lx_srcpos        : source_position,
                     lx_comments      : comments,
                     lx_symrep        : symbol_rep;
out_id =>            sm_obj_type      : TYPE_SPEC,
                     sm_first         : DEF_OCCURRENCE;
```

```
-- 6.3  Subprogram Bodies

-- Syntax 6.3
--   subprogram_body ::=
--        subprogram_specification is
--            [declarative_part]
--        begin
--            sequence_of_statements
--      [exception
--             exception_handler
--            {exception_handler}]
--        end [designator];
--
```

```
BLOCK_STUB ::=          block;

subprogram_body =>      as_designator       : DESIGNATOR,      -- one of 'proc_id',
                                                      -- 'function_id' or 'def_op'
                        as_header           : HEADER,
                        as_block_stub       : BLOCK_STUB;
subprogram_body =>      lx_srcpos           : source_position,
                        lx_comments         : comments;
```

-- 6.4 Subprogram Calls

```
-- Syntax 6.4
--  procedure_call_statement ::=
--     procedure_name [actual_parameter_part];
--
--  function_call ::=
--     function_name [actual_parameter_part]
--
--  actual_parameter_part ::=
--     (parameter_association {, parameter_association})
--
--  parameter_association ::=
--     [formal_parameter =>] actual_parameter
--
--  formal_parameter ::= parameter_simple_name
--
--  actual_parameter ::=
--     expression | variable_name | type_mark(variable_name)
--
```

```
procedure_call =>      as_name               : NAME,
                       as_param_assoc_s      : PARAM_ASSOC_S;
procedure_call =>      lx_srcpos             : source_position,
                       lx_comments           : comments;
procedure_call =>      sm_normalized_param_s        :EXP_S;

function_call =>       as_name               : NAME,
                       as_param_assoc_s      : PARAM_ASSOC_S;
function_call =>       lx_srcpos             : source_position,
                       lx_comments           : comments;
function_call =>       sm_exp_type           : TYPE_SPEC,
                       sm_value              : value,
                       sm_normalized_param_s        : EXP_S,
                       lx_prefix             : Boolean;

PARAM_ASSOC ::=        EXP | assoc;

assoc =>               as_designator         : DESIGNATOR,
                       as_actual             : ACTUAL;
assoc =>               lx_srcpos             : source_position,
                       lx_comments           : comments;

ACTUAL ::=             EXP;
```

```
-- 7.  Packages
--
-- 7.1  Package Structure

-- Syntax 7.1.A
--  package_declaration ::= package_specification;
--

    package_decl =>        as_id             : ID,      -- always 'package_id'
                           as_package_def    : PACKAGE_DEF;
    package_decl =>        lx_srcpos         : source_position,
                           lx_comments       : comments;

    DEF_ID ::=             package_id;

    package_id =>          lx_srcpos         : source_position,
                           lx_comments       : comments,
                           lx_symrep         : symbol_rep;
    package_id =>          sm_spec           : PACKAGE_SPEC,
                           sm_body           : PACK_BODY_DESC,
                           sm_address        : EXP_VOID,
                           sm_stub           : DEF_OCCURRENCE,
                           sm_first          : DEF_OCCURRENCE;

    PACK_BODY_DESC::=      block | stub | rename | instantiation | void;


-- Syntax 7.1.B
--  package_specification ::=
--     package identifier is
--          {basic_declarative_item}
--     [private
--          {basic_declarative_item}]
--     end [package_simple_name]
--


    PACKAGE_SPEC ::=       package_spec;
    PACKAGE_DEF ::=        package_spec;

    package_spec =>        as_decl_s1        : DECL_S,     -- visible declarations
                           as_decl_s2        : DECL_S;     -- private declarations
    package_spec =>        lx_srcpos         : source_position,
                           lx_comments       : comments;

    DECL_S ::=             decl_s;

    decl_s =>              as_list           : Seq Of DECL;
    decl_s =>              lx_srcpos         : source_position,
                           lx_comments       : comments;
```

ADA Section 6.4

```
-- Syntax 7.1.C
--  package_body ::=
--      package body package_simple_name is
--          [declarative_part]
--     [begin
--          sequence_of_statements
--     [exception
--          exception_handler
--          {exception_handler}]]
--      end [package_simple_name];
--

    package_body =>         as_id               : ID,        -- always 'package_id'
                            as_block_stub       : BLOCK_STUB;
    package_body =>         lx_srcpos           : source_position,
                            lx_comments         : comments;
```

-- 7.4 Private Type and Deferred Constant Declarations

```
-- Syntax 7.4.A
--  private_type_declaration ::=
--      type identifier [discriminant_part] is [limited] private;
--

    TYPE_SPEC ::=           private;
    TYPE_SPEC ::=           l_private;

    private =>              lx_srcpos           : source_position,
                            lx_comments         : comments;
    private =>              sm_discriminants    : DSCRMT_VAR_S;
    l_private =>            lx_srcpos           : source_position,
                            lx_comments         : comments;
    l_private =>            sm_discriminants    : DSCRMT_VAR_S;

    DEF_ID ::=              private_type_id | l_private_type_id;

    private_type_id =>      lx_srcpos           : source_position,
                            lx_comments         : comments,
                            lx_symrep           : symbol_rep;
    private_type_id =>      sm_type_spec        : TYPE_SPEC;
                                                    -- Refers to the complete
                                                    -- type specification of the
                                                    -- private type.
                                                    -- See 3.4.2.4 of rationale.

    l_private_type_id =>    lx_srcpos           : source_position,
                            lx_comments         : comments,
                            lx_symrep           : symbol_rep;
    l_private_type_id =>    sm_type_spec        : TYPE_SPEC;
                                                    -- Refers to the complete
                                                    -- type specification of the
                                                    -- limited private type.
                                                    -- See 3.4.2.4 of rationale.


-- Syntax 7.4.B
--  deferred_constant_declaration ::=
--      identifier_list : constant type_mark;

    deferred_constant =>    as_id_s             : ID_S,     -- sequence of 'const_id'
                            as_name             : NAME;
    deferred_constant =>    lx_srcpos           : source_position,
                            lx_comments         : comments;
```

```
-- 8.  Visibility Rules
--
-- 8.4  Use Clauses

-- Syntax 8.4
--   use_clause ::= use package_name {, package_name};
--

    use =>                as_list              : Seq Of NAME;
    use =>                lx_srcpos            : source_position,
                          lx_comments          : comments;


-- 8.5  Renaming Declarations

-- Syntax 8.5
--   renaming_declaration ::=
--           identifier : type_mark      renames object_name;
--        | identifier : exception       renames exception_name;
--        | package identifier           renames package_name;
--        | subprogram_specification     renames subprogram_or_entry_name;
--

    -- See Section 3.7 of rationale for discussion of renaming

    OBJECT_DEF ::=        rename;
    EXCEPTION_DEF ::=     rename;
    PACKAGE_DEF ::=       rename;
    SUBPROGRAM_DEF ::=    rename;


    rename =>             as_name              : NAME;
    rename =>             lx_srcpos            : source_position,
                          lx_comments          : comments;
```

```
-- 9.  Tasks
--
-- 9.1  Task Specifications and Task Bodies

-- Syntax 9.1.A
--  task_declaration ::= task_specification;
--
--  task_specification ::=
--     task [type] identifier [is
--          {entry_declaration}
--          {representation_clause}
--     end [task_simple_name]]
--

    -- see 3.3 for task type declaration

    TASK_DEF ::=            task_spec;

    task_decl =>            as_id            : ID,      --always a var_id
                            as_task_def      : TASK_DEF;
    task_decl =>            lx_srcpos        : source_position,
                            lx_comments      : comments;

    TYPE_SPEC ::=           task_spec;

    task_spec =>            as_decl_s        : DECL_S;
    task_spec =>            lx_srcpos        : source_position,
                            lx_comments      : comments;
    task_spec =>            sm_body          : BLOCK_STUB_VOID,      -- Void only
                                                        -- in the presence
                                                        -- of separate compilation.
                                                        -- See 3.5.5 of rationale.
                            sm_address       : EXP_VOID,
                            sm_storage_size  : EXP_VOID;

    BLOCK_STUB_VOID ::=  block | stub | void;


-- Syntax 9.1.B
--  task_body ::=
--      task body task_simple_name is
--          [declarative_part]
--      begin
--          sequence_of_statements
--     [exception
--          exception_handler
--          {exception_handler}]
--      end [task_simple_name];
--


    task_body =>            as_id            : ID,      -- always 'task_body_id'
                            as_block_stub    : BLOCK_STUB;
    task_body =>            lx_srcpos        : source_position,
                            lx_comments      : comments;

    DEF_ID ::=              task_body_id;

    task_body_id =>         lx_srcpos        : source_position,
                            lx_comments      : comments,
                            lx_symrep        : symbol_rep;
    task_body_id =>         sm_type_spec     : TYPE_SPEC,
                            sm_body          : BLOCK_STUB_VOID,
                            sm_first         : DEF_OCCURRENCE,
                            sm_stub          : DEF_OCCURRENCE;
```

```
-- 9.5  Entries, Entry Calls and Accept Statements

-- Syntax 9.5.A
--   entry_declaration ::=
--      entry identifier [(discrete_range)] [formal_part];
--

    -- entry uses subprogram_decl, see 6.1

    HEADER ::=              entry;
    DSCRT_RANGE_VOID ::=    DSCRT_RANGE | void;

    entry =>                as_dscrt_range_void  : DSCRT_RANGE_VOID,
                            as_param_s           : PARAM_S;
    entry =>                lx_srcpos            : source_position,
                            lx_comments          : comments;

    DEF_ID ::=              entry_id;

    entry_id =>             lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    entry_id =>             sm_spec              : HEADER,
                            sm_address           : EXP_VOID;

-- Syntax 9.5.B
--   entry_call_statement ::= entry_name [actual_parameter_part];
--

    entry_call =>           as_name              : NAME, -- indexed when entry of family
                            as_param_assoc_s     : PARAM_ASSOC_S;
    entry_call =>           lx_srcpos            : source_position,
                            lx_comments          : comments;
    entry_call =>           sm_normalized_param_s      :EXP_S;

-- Syntax 9.5.C
--   accept_statement ::=
--      accept entry_simple_name [(entry_index)] [formal_part] [do
--           sequence_of_statements
--      end [entry_simple_name]];
--
--   entry_index ::= expression
--

    accept =>               as_name              : NAME,
                            as_param_s           : PARAM_S,
                            as_stm_s             : STM_S;
    accept =>               lx_srcpos            : source_position,
                            lx_comments          : comments;

-- 9.6  Delay Statements, Duration and Time

-- Syntax 9.6
--   delay_statement ::= delay simple_expression;
--

    delay =>                as_exp               : EXP;
    delay =>                lx_srcpos            : source_position,
                            lx_comments          : comments;
```

-- 9.7 Select Statements

```
-- Syntax 9.7
--   select_statement ::= selective_wait
--       | conditional_entry_call | timed_entry_call
--


    -- see below
```

-- 9.7.1 Selective Waits

```
-- Syntax 9.7.1.A
--   selective_wait ::=
--        select
--            select_alternative
--       {or
--            select_alternative}
--       [else
--            sequence_of_statements]
--        end select;
--


    select =>                as_select_clause_s   : SELECT_CLAUSE_S,
                             as_stm_s             : STM_S;
    select =>                lx_srcpos            : source_position,
                             lx_comments          : comments;

    SELECT_CLAUSE_S ::=      select_clause_s;
    select_clause_s =>       as_list              : Seq Of SELECT_CLAUSE;
    select_clause_s =>       lx_srcpos            : source_position,
                             lx_comments          : comments;


-- Syntax 9.7.1.B
--   selective_alternative ::=
--       [when condition =>]
--            selective_wait_alternative
--
--   selective_wait_alternative ::= accept_alternative
--       | delay_alternative | terminate_alternative
--
--   accept_alternative ::= accept_statement [sequence_of_statements]
--
--   delay_alternative ::= delay_statement  [sequence_of_statements]
--
--   terminate_alternative ::= terminate;
--


    SELECT_CLAUSE ::=        select_clause;
    SELECT_CLAUSE ::=        pragma;              -- pragma allowed where alternative allowed

    select_clause =>         as_exp_void          : EXP_VOID,
                             as_stm_s             : STM_S; -- first stm is accept or delay
    select_clause =>         lx_srcpos            : source_position,
                             lx_comments          : comments;

    STM ::=                  terminate;

    terminate =>             lx_srcpos            : source_position,
                             lx_comments          : comments;
```

```
-- 9.7.2  Conditional Entry Calls

-- Syntax 9.7.2
--  conditional_entry_call ::=
--      select
--            entry_call_statement
--           [sequence_of_statements]
--      else
--           sequence_of_statements
--      end select;
--

   cond_entry =>         as_stm_s1          : STM_S,-- first stm is entry_call
                         as_stm_s2          : STM_S;
   cond_entry =>         lx_srcpos          : source_position,
                         lx_comments        : comments;

-- 9.7.3  Timed Entry Calls

-- Syntax 9.7.3
--  timed_entry_call ::=
--      select
--            entry_call_statement
--           [sequence_of_statements]
--      or
--            delay_alternative
--      end select;
--

   timed_entry =>        as_stm_s1          : STM_S,-- first stm is entry_call
                         as_stm_s2          : STM_S;-- first stm is delay
   timed_entry =>        lx_srcpos          : source_position,
                         lx_comments        : comments;

-- 9.10  Abort Statements

-- Syntax 9.10
--  abort_statement ::= abort task_name {, task_name};
--

   NAME_S ::=            name_s;

   name_s =>             as_list            : Seq Of NAME;
   name_s =>             lx_srcpos          : source_position,
                         lx_comments        : comments;

   abort =>              as_name_s          : NAME_S;
   abort =>              lx_srcpos          : source_position,
                         lx_comments        : comments;
```

```
-- 10.  Program Structure and Compilation Issues
--
-- 10.1  Compilation Units - Library Units

-- Syntax 10.1.A
--   compilation ::= {compilation_unit}
--

    COMPILATION ::=         compilation;

    compilation =>          as_list               : Seq Of COMP_UNIT;
    compilation =>          lx_srcpos             : source_position,
                            lx_comments           : comments;


-- Syntax 10.1.B
--   compilation_unit ::=
--      context_clause library_unit | context_clause secondary_unit
--
--   library_unit ::=
--         subprogram_declaration | package_declaration
--       | generic_declaration    | generic_instantiation
--       | subprogram_body
--
--   secondary_unit ::= library_unit_body | subunit
--
--   library_unit_body ::= subprogram_body | package_body
--


    COMP_UNIT ::=           comp_unit;
    UNIT_BODY ::=           package_body | package_decl | subunit | generic
                             | subprogram_body | subprogram_decl | void;
    -- UNIT_BODY is void only when comp_unit consists of only pragmas

    PRAGMA_S ::=            pragma_s;

    pragma_s =>             as_list               : Seq Of PRAGMA;
    pragma_s =>             lx_srcpos             : source_position,
                            lx_comments           : comments;

    comp_unit =>            as_context            : CONTEXT,
                            as_unit_body          : UNIT_BODY,
                            as_pragma_s           : PRAGMA_S; -- extension to FD.
    comp_unit =>            lx_srcpos             : source_position,
                            lx_comments           : comments;

    CONTEXT_ELEM ::=        pragma;               -- pragma allowed in clause

-- Context Clauses - With Clauses

-- Syntax 10.1.1.A
--   context_clause ::= {with_clause {use_clause}}
--


    CONTEXT_ELEM ::=        use;
    CONTEXT ::=             context;

    context =>              as_list               : Seq Of CONTEXT_ELEM;
    context =>              lx_srcpos             : source_position,
                            lx_comments           : comments;
```

```
-- Syntax 10.1.1.B
--   with_clause ::= with unit_simple_name {, unit_simple_name};
--

    CONTEXT_ELEM ::=        with;

    with =>                 as_list              : Seq Of NAME;
    with =>                 lx_srcpos            : source_position,
                            lx_comments          : comments;
```

-- 10.2 Subunits of Compilation Units

```
-- Syntax 10.2.A
--   subunit ::=
--       separate (parent_unit_name) proper_body
--

    subunit =>              as_name              : NAME,
                            as_subunit_body      : SUBUNIT_BODY;
    subunit =>              lx_srcpos            : source_position,
                            lx_comments          : comments;

    SUBUNIT_BODY ::=        subprogram_body | package_body | task_body;


-- Syntax 10.2.B
--   body_stub ::=
--         subprogram_specification is separate;
--       | package body package_simple_name is separate;
--       | task body task_simple_name is separate;
--

    BLOCK_STUB ::=          stub;

    stub =>                 lx_srcpos            : source_position,
                            lx_comments          : comments;
```

-- 11. Exceptions
--
-- 11.1 Exception Declarations

```
-- Syntax 11.1
--   exception_declaration ::= identifier_list : exception;
--

    EXCEPTION_DEF ::=       void;

    exception =>            as_id_s              : ID_S,    -- 'exception_id' sequence
                            as_exception_def     : EXCEPTION_DEF;
    exception =>            lx_srcpos            : source_position,
                            lx_comments          : comments;

    DEF_ID ::=              exception_id;

    exception_id =>         lx_srcpos            : source_position,
                            lx_comments          : comments,
                            lx_symrep            : symbol_rep;
    exception_id =>         sm_exception_def     : EXCEPTION_DEF;
```

-- 11.2 Exception Handlers

```
-- Syntax 11.2
--   exception_handler ::=
--      when exception_choice {| exception_choice} =>
--          sequence_of_statements
--
--   exception_choice ::= exception_name | others
--


    -- see 5.4, 5.6, 3.7.3.B
```

-- 11.3 Raise Statements

```
-- Syntax 11.3
--   raise_statement ::= raise [exception_name];
--


    raise =>            as_name_void        : NAME_VOID;
    raise =>            lx_srcpos           : source_position,
                        lx_comments         : comments;
```

-- 12. Generic Program Units

-- 12.1 Generic Declarations

```
-- Syntax 12.1.A
--   generic_declaration ::= generic_specification;
--
--   generic_specification ::=
--        generic_formal_part subprogram_specification
--      | generic_formal_part package_specification
--


    GENERIC_HEADER ::=  procedure | function | package_spec;

    generic =>          as_id               : ID,      -- 'generic_id'
                        as_generic_param_s  : GENERIC_PARAM_S,
                        as_generic_header   : GENERIC_HEADER;
    generic =>          lx_srcpos           : source_position,
                        lx_comments         : comments;

    DEF_ID ::=          generic_id;

    generic_id =>       lx_symrep           : symbol_rep,
                        lx_srcpos           : source_position,
                        lx_comments         : comments;
    generic_id =>       sm_generic_param_s  : GENERIC_PARAM_S,
                        sm_spec             : GENERIC_HEADER,
                        sm_body             : BLOCK_STUB_VOID,
                        sm_first            : DEF_OCCURRENCE,
                        sm_stub             : DEF_OCCURRENCE;
```

```
-- Syntax 12.1.B
--   generic_formal_part ::= generic {generic_parameter_declaration}
--

   GENERIC_PARAM_S ::=  generic_param_s;

   generic_param_s =>   as_list               : Seq Of GENERIC_PARAM;
   generic_param_s =>   lx_srcpos             : source_position,
                        lx_comments           : comments;


-- Syntax 12.1.C
--   generic_parameter_declaration ::=
--         identifier_list : [in [out]] type_mark [:= expression];
--       | type identifier is generic_type_definition;
--       | private_type_declaration
--       | with subprogram_specification [is name];
--       | with subprogram_specification [is <>];
--


   GENERIC_PARAM ::=    in | in_out | type | subprogram_decl;

   SUBPROGRAM_DEF ::=   FORMAL_SUBPROG_DEF;

   FORMAL_SUBPROG_DEF ::=                     NAME | box | no_default;

   box =>               lx_srcpos             : source_position,
                        lx_comments           : comments;
   no_default =>        lx_srcpos             : source_position,
                        lx_comments           : comments;


-- Syntax 12.1.D
--   generic_type_definition ::=
--         (<>) | range <> | digits <> | delta <>
--       | array_type_definition | access_type_definition
--


   TYPE_SPEC ::=        FORMAL_TYPE_SPEC;
   FORMAL_TYPE_SPEC ::= formal_dscrt          -- (<>)
                      | formal_integer        -- range <>
                      | formal_fixed          -- delta <>
                      | formal_float;         -- digits <>

   formal_dscrt =>      lx_srcpos             : source_position,
                        lx_comments           : comments;
   formal_fixed =>      lx_srcpos             : source_position,
                        lx_comments           : comments;
   formal_float =>      lx_srcpos             : source_position,
                        lx_comments           : comments;
   formal_integer =>    lx_srcpos             : source_position,
                        lx_comments           : comments;
```

-- 12.3 Generic Instantiation

```
-- Syntax 12.3.A
--   generic_instantiation ::=
--        package identifier is
--             new generic_package_name [generic_actual_part];
--      | procedure identifier is
--             new generic_procedure_name [generic_actual_part];
--      | function identifier is
--             new generic_function_name [generic_actual_part];
--
--   generic_actual_part ::=
--      (generic_association {, generic_association})
--

    -- See 3.6 of rationale for discussion of instantiation

    SUBPROGRAM_DEF ::=  instantiation;

    PACKAGE_DEF ::=     instantiation;

    GENERIC_ASSOC_S ::= generic_assoc_s;

    generic_assoc_s =>  as_list              : Seq Of GENERIC_ASSOC;
    generic_assoc_s =>  lx_srcpos            : source_position,
                        lx_comments          : comments;

    instantiation =>    as_name              : NAME,
                        as_generic_assoc_s   : GENERIC_ASSOC_S;
    instantiation =>    lx_srcpos            : source_position,
                        lx_comments          : comments;
    instantiation =>    sm_decl_s            : DECL_S;


-- Syntax 12.3.B
--   generic_association ::=
--      [generic_formal_parameter =>] generic_actual_parameter
--
--   generic_formal_parameter ::= parameter_simple_name | operator_symbol
--


    GENERIC_ASSOC ::=   assoc;


-- Syntax 12.3.C
--   generic_actual_parameter ::= expression | variable_name
--      | subprogram_name | entry_name | type_mark
--


    GENERIC_ASSOC ::=   ACTUAL;
```

```
-- 13. Representation Clauses and
--
-- Implementation Dependent Features
--
-- 13.1  Representation Clauses

-- Syntax 13.1
--  representation_clause ::=
--       type_representation_clause | address_clause
--
--  type_representation_clause ::= length_clause
--     | enumeration_representation_clause | record_representation_clause
--

    REP ::=            simple_rep       -- length_clause and
                                        -- enumeration_representation_clause (13.2)
                     | address          -- address_clause (13.5)
                     | record_rep;      -- record_representation_clause (13.4)

-- 13.2 Length Clause
-- 13.3 Enumeration Representation Clauses

-- Syntax 13.2
--  length_clause ::= for attribute use simple_expression;

-- Syntax 13.3
--  enumeration_representation_clause ::=
--       for type_simple_name use aggregate;
--

    simple_rep =>      as_name          : NAME,
                       as_exp           : EXP;
    simple_rep =>      lx_srcpos        : source_position,
                       lx_comments      : comments;

-- 13.4  Record Representation Clauses

-- Syntax 13.4.A
--  record_representation_clause ::=
--     for type_simple_name use
--         record [alignment_clause]
--             {component_clause}
--         end record;
--
--  alignment_clause ::= at mod static_simple_expression;
--

    ALIGNMENT ::=      alignment;

    alignment =>       as_pragma_s      : PRAGMA_S, -- pragma allowed in clause
                       as_exp_void      : EXP_VOID;

    record_rep =>      as_name          : NAME,
                       as_alignment     : ALIGNMENT,
                       as_comp_rep_s    : COMP_REP_S;
    record_rep =>      lx_srcpos        : source_position,
                       lx_comments      : comments;
```

```
-- Syntax 13.4.B
--  component_clause ::=
--     component_simple_name at static_simple_expression range static_range;
--

    COMP_REP_S ::=       comp_rep_s;
    COMP_REP ::=         comp_rep;
    COMP_REP ::=         pragma;            -- pragma allowed in clause

    comp_rep_s =>        as_list            : Seq Of COMP_REP;
    comp_rep_s =>        lx_srcpos          : source_position,
                         lx_comments        : comments;

    comp_rep =>          as_name            : NAME,
                         as_exp             : EXP,
                         as_range           : RANGE;
    comp_rep =>          lx_srcpos          : source_position,
                         lx_comments        : comments;
```

-- 13.5 Address Clauses

```
-- Syntax 13.5
--  address_clause ::= for simple_name use at simple_expression;
--

    address =>           as_name            : NAME,
                         as_exp             : EXP;
    address =>           lx_srcpos          : source_position,
                         lx_comments        : comments;
```

-- 13.8 Machine Code Insertions

```
-- Syntax 13.8
--  code_statement ::= type_mark'record_aggregate;
--

    code =>              as_name            : NAME,
                         as_exp             : EXP;
    code =>              lx_srcpos          : source_position,
                         lx_comments        : comments;
```

-- 14.0 Input-Output

```
--
-- I/O procedure calls are not specially handled. They are
-- represented by procedure or function calls (see 6.4).
```

```
-- Predefined Diana Environment
--
--
-- see Appendix I of this manual
--

    DEF_ID ::=              attr_id | pragma_id | ARGUMENT;
    ARGUMENT ::=            argument_id;

    attr_id =>              lx_symrep          : symbol_rep;

    TYPE_SPEC ::=           universal_integer | universal_fixed | universal_real;

    universal_integer =>    ;
    universal_fixed =>      ;
    universal_real =>       ;

    argument_id =>          lx_symrep          : symbol_rep;

    pragma_id =>            as_list            : Seq Of ARGUMENT;
    pragma_id =>            lx_symrep          : symbol_rep;

  End
```

```
Structure Diana_Concrete
Refines Diana is
--
--
--
--   Refined Diana Specification
--   ===========================
--
--
--      Version of 11 February 1983
--
--

For source_position     Use USERPK.SOURCE_POSITION;
                        -- defines source position in original
                        -- source program. used for error messages.
For symbol_rep          Use USERPK.SYMBOL_REP;
                        -- representation of identifiers,
                        -- strings and characters
For value               Use USERPK.MACHINE_VALUE;
                        -- implementation defined
                        -- gives value of an expression.
                        -- can indicate that no value is computed.
For operator            Use USERPK.OPERATOR;
                        -- enumeration type for all operators
For number_rep          Use USERPK.NUMBER_REP;
                        -- representation of numeric literals
For comments            Use USERPK.COMMENTS;
                        -- representation of comments from source program


--
--      This defines the external representations
--
For symbol_rep          Use External String;
                        -- the external representation of
                        -- symbol_rep uses IDL basic type string.
For number_rep          Use External String;
                        -- the external representation of
                        -- number_rep uses IDL basic type string.
For operator            Use  External OP_CLASS;
                        -- the external representation of operator
                        -- uses the private type OP_CLASS
For value               Use  External VAL_CLASS;
                        -- the external representation of values
                        -- uses the private type VAL_CLASS
```

```
--
-- OP_CLASS is an enumeration class that defines the Ada operators
--
-- Syntax 4.5
--   logical_operator ::= and | or | xor
--
--   relational_operator ::=  = | /= | < | <= | > | >=
--
--   adding_operator ::= + | - | &
--
--   unary_operator ::= + | -
--
--   multiplying_operator ::= * | / | mod | rem
--
--   highest_precedence_operator ::= ** | abs | not
--

    OP_CLASS ::=          and                  -- and
                        | or                   -- or
                        | xor                  -- xor
                        | eq                   -- =
                        | ne                   -- /=
                        | lt                   -- <
                        | le                   -- <=
                        | gt                   -- >
                        | ge                   -- >=
                        | plus                 -- +
                        | minus                -- -
                        | cat                  -- &
                        | unary_plus           -- +
                        | unary_minus          -- -
                        | abs                  -- abs
                        | not                  -- not
                        | mult                 -- *
                        | div                  -- /
                        | mod                  -- mod
                        | rem                  -- rem
                        | exp;                 -- **

     and => ;            or => ;             xor => ;          eq => ;
     ne => ;             lt => ;             le => ;           gt => ;
     ge => ;             plus => ;           minus => ;        cat => ;
     unary_plus => ;     unary_minus => ;    abs => ;          not => ;
     mult => ;           div => ;            mod => ;          rem => ;
     exp => ;

--
-- VAL_CLASS is a class that defines the possible Diana values
--

    VAL_CLASS ::=         no_value | string_value | bool_value |
                          int_value | real_value;

    no_value =>           ;                    -- no value has been computed
    string_value =>       str_val            : String; -- character and string
    bool_value =>         boo_val            : Boolean;     -- boolean value
    int_value =>          int_val            : Integer;     -- integer value
    real_value =>         rtn_val            : Rational;    -- real and fixed values

End
```

CHAPTER 3
RATIONALE

The design of DIANA is based on the principles listed in Section 1.1. Unfortunately these principles are *not* always compatible with each other and with ADA. Under some circumstances it was necessary to deviate from them, albeit in minor ways. The main purpose of this chapter is to clarify the DIANA approach and to give reasons for our compromise decisions.

An important principle in the design of DIANA was to adhere to the *Formal Definition* of ADA (AFD), and in particular, to the abstract syntax defined there. The first section below compares DIANA trees with those of the Abstract Syntax and shows the transformations from the DIANA form back to that given in the AFD. The second section describes the effects of separate compilation on DIANA. The third section discusses the DIANA approach to the notion of a dictionary or symbol table. In the fourth section we discuss an important output of the semantic analyzer—the type information about objects. We point out special situations and solutions which may not be obvious from the definition given in the last chapter. The fifth section discusses another principle that it was not possible to apply consistently—the requirement that there be a single definition for each entity. Here the language, and especially its separate compilation facility, impose a compromise on DIANA. The sixth and seventh sections discuss the special problems of instantiations and renaming. The eighth section deals with implementation dependent attribute types that are introduced in DIANA in order to avoid constraining an implementation. The ninth section discusses the notions of equality and assignment for attributes. A summary of the non-structural attributes closes the chapter.

This chapter contains a number of examples where the structure of DIANA trees is given in a graphical manner to illustrate the relations between attribute values and nodes. To emphasize the important points, we show only those parts of the structure which are of interest for the particular example. Thus, a subtree is sometimes replaced by the string which it represents or by ellipses if it is not important. If attributes are attached to a node, then the kind of the node and the attributes of interest are enclosed in a box. It is our intention that these figures capture only the essential information for the purpose at hand and hence suppress unnecessary detail; they should not be viewed as complete.

3.1. Comparison with the Abstract Syntax Tree

In this section we show that the Abstract Syntax Trees used in the AFD [6] and the DIANA trees (with only structural attributes) are equivalent. This equivalence is useful for the description of the semantics of a DIANA tree: we simply inherit the semantics from the AFD. Further, it enforces standardization of the abstract syntax representation of programs. Since, however, it was necessary to deviate from the AFD in minor ways, we list these deviations and point out the reasons why they are necessary: we also indicate how the Abstract Syntax Tree can be reconstructed from the DIANA tree.

We recognize that the ADA AFD is based on the 1980 revised ADA Language Reference Manual [7] and does not reflect changes made to the syntax in the 1982 reference manual. This issue is addressed in Section 3.1.5.

3.1.1. Semantic Distinctions of Constructs

Several nodes in DIANA have no counterpart in the Abstract Syntax of the AFD. They are introduced in cases where a single construct in the AFD may have several distinct semantic meanings. Different nodes allow us to attach appropriate semantic attributes to each. In all such cases the name of the original construct is extended with prefixes which denote the distinction. The largest number of splits has been made for the id-construct: we not only distinguish between a defining occurrence and a used occurrence of an identifier, but also between the kinds of the items denoted by it. For example,

const_id is a node which can appear in a constant declaration to define a constant object. If such an object is referenced by an identifier in an expression, the construct

used_object_id is used. The semantic attributes of both constructs can be found in the DIANA definition.

Note that the attributes of these two types of '_id' nodes are disjoint and that their union contains all the information needed.

The original Abstract Syntax Tree can easily be reconstructed by omitting the prefix of these nodes. It should be noted that no tree transformation is necessary, since the structure of the new DIANA nodes is the same as that of their counterparts in the Abstract Syntax.

3.1.2. Additional Concepts

There are nodes introduced in DIANA which are used to deal with issues that are not considered in the AFD. They are used to represent pragmas and parentheses in expressions. If the nodes for parentheses and pragmas are removed from the tree, the original Abstract Syntax structure is restored.

Under some circumstances parentheses have a semantic effect in ADA. Consider the following examples:

```
F( (A) )            -- Parameter cannot be in or in out
A + (B + C)         -- Parentheses force the grouping
(A + B) * C         -- Parentheses force the proper parse
```

In each of these cases the parentheses have a semantic effect. In addition, the ADA conformance rules (see Section 6.3.1 of the ADA LRM [8]) require that parentheses be preserved in order to check that subprogram specifications match. DIANA requires that all parentheses in the original ADA source are preserved through the use of **parenthesized** nodes. See Section 1.1.3.

Pragmas may carry the commands given by the user to other compiler modules after semantic analysis and must be preserved. Since pragmas may occur in so many places in ADA (see Section 2.8 of the ADA LRM [8]), many DIANA classes were expanded to allow pragmas. This does not affect the structure of the abstract syntax tree. However, the presence of pragmas also caused us to change the structure of the **comp_unit** node of the abstract syntax. Pragmas can be given for a compilation unit and are therefore represented together with the corresponding node. The **comp_unit** node now has three children:

```
comp_unit =>        context       : CONTEXT,
                    unit_body     : UNIT_BODY,
                    pragma_s      : PRAGMA_S ;
```

From the abstract datatype viewpoint, DIANA has merely added one additional selector. The original selectors of the AFD are retained unchanged.

3.1.3. Tree Normalizations

The AFD uses various normalizations of the tree. Most, but not all, of them are also imposed by DIANA. Those which are not performed in DIANA were elided because after such normalizations it is difficult, and sometimes impossible, to reconstruct the source text.

We do not follow the AFD in normalizing anonymous types. The AFD proposes that all anonymous types be replaced by type marks and have an

explicit declaration just before their original appearance. This tree transformation is not required by DIANA. For example, the declaration of a task object does not require a declaration of an anonymous task type to be placed in the DIANA tree before the task object.

We do not normalize parameter associations. In the AFD, all subprogram calls have their parameter sequences normalized to the *named association form*. DIANA leaves positional parameters as the user wrote them and avoids filling in default parameters. (DIANA does have a semantic attribute for subprogram calls that normalizes parameter sequences and fills in default parameters, but semantic attributes are not represented in the Abstract Syntax Tree).

All other normalizations in the AFD (*e.g.*, treating built-in operators as function calls) *are* imposed by DIANA. The impact of these normalizations on reconstruction of the original source program from the DIANA tree is discussed in Appendix III. The normalizations which are not assumed by DIANA must be done to get the Abstract Syntax Tree; the AFD defines how these are done.

3.1.4. Tree Transformation According to the Formal Definition

Some ambiguities of the concrete syntax cannot be resolved by the parser, but must be removed during semantic analysis. For example, the Abstract Syntax contains an apply construct, covering indexed expressions, calls, conversions, and slices. In most cases semantic analysis merely has to rename the node to encode the nature of the construct; there are no structural differences. The result of this process is assumed in DIANA as well as in the AFD (See Appendix II). It should be noted that one possibility requires a structural transformation of the tree, namely when an **apply** node has to be changed into a call to a parameterless entry family member. Figure 3-1 illustrates this case. All these changes are in accordance with the AFD and require no actions to reconstruct the Abstract Syntax Tree.

3.1.5. Changes to the AST

The majority of the changes in ADA syntax have not produced a change in the structure of the Abstract Syntax Tree. For example, the change in syntax that requires the result subtype of a function to be specified by a type mark instead of a subtype indication has allowed DIANA to use a NAME as a child of the **function** instead of a CONSTRAINED node. This does not affect the structure in the sense that the number of children that the **function** node has has not

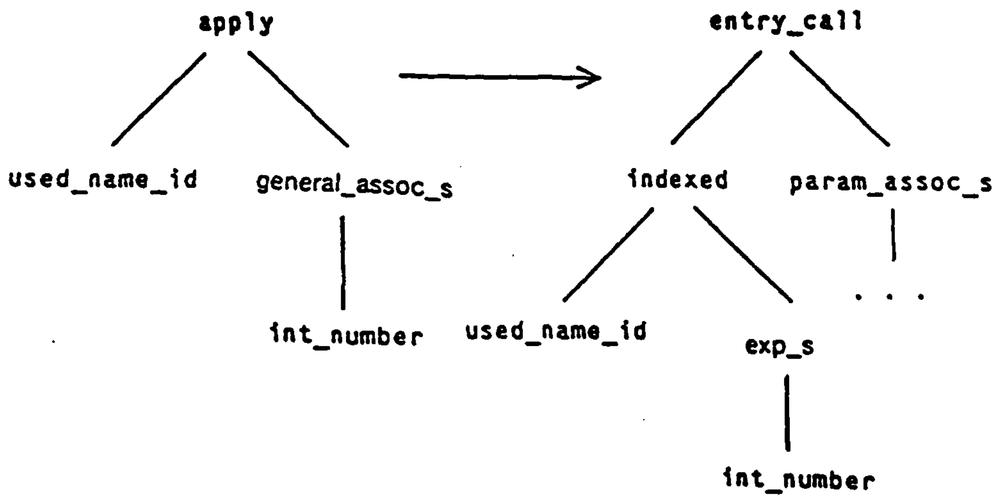


Figure 3-1: Example of a Necessary Tree Transformation

changed. One node has been changed structurally, the **allocator** node, which has been changed to have only one child, *as_exp_constrained*, instead of the two children specified in the Abstract Syntax Tree defined in the AFD.

Two DIANA nodes have been introduced to consistently represent the changes to ADA syntax. The discriminant specification requires a type mark instead of a subtype indication. The Abstract Syntax Tree uses a var node to represent both discriminant specifications and variable declarations. DIANA uses a separate node, **dscrmt_var**, to represent the discriminant specification. Similarly, a deferred constant declaration differs from a full constant declaration in that it requires a type mark instead of a subtype indication. Both are represented by a **constant** node in the Abstract Syntax Tree. DIANA represents the deferred constant declaration with the **deferred_constant** node.

3.2. Consequences of Separate Compilation

The separate compilation facility of ADA affects the intermediate representation of programs. The Front End must be able to use the intermediate representation of a previously compiled unit again. Further, the Front End may not have complete information about a program unit.

The design of DIANA carefully avoids constraints on a separate compilation system, aside from those implied directly by the ADA language. The

design can be extended to cover the full APSE requirements. We have taken special care that several versions of a unit body can exist corresponding to a single specification, that simultaneous compilation within the same project is possible, and that units of other libraries can be used effectively [5].

The basic decision which makes these facilities implementable is to forbid forward references: this decision is explained in the next section. We then point out some limitations imposed on the Front End by the separate compilation facility.

3.2.1. Forward References

The basic principle of DIANA that there is a single definition point for each ADA entity conflicts with those ADA facilities that have more than one declaration point. In these cases, DIANA restricts the attribute values of all the defining occurrences to be identical (see Section 3.5). In the presence of separate compilation, the requirement that the values of the attributes at all defining occurrences are the same can only be met temporarily. The forward references (*sm_body*) assumed by DIANA are void in these cases. The reasons for this approach are:

- A unit can be used even when the corresponding body is not yet compiled. In this case, the forward reference must have the value void since the entity does not exist.
- Updating a DIANA representation would require write access to a file which may cause synchronization problems (see [5]).
- A library system may allow for several versions of bodies for the same specification. If we were to update an attribute, we would overwrite its previous value. Moreover, we believe that the maintenance of different versions should be part of the library system and should not influence the intermediate representation.

3.2.2. Separately Compiled Generic Bodies

The ADA separate compilation facility does not impose a total order on compilations. It is possible to use a unit whose body has not yet been compiled, provided that its specification has been compiled. This procedure does not normally cause a problem, since the specification usually contains all the information needed to use a unit.

However, a generic unit can be instantiated regardless of whether the generic

body has been compiled. Thus, in many cases the Front End cannot instantiate the body at the time it compiles an instantiation. It would be possible to keep track of the instantiations and compile them once the body becomes available. But this method would imply that already-stored intermediate representations have to be modified. After such an update, existing references to the updated unit might be invalid.

DIANA assumes that only the specification is instantiated (see Section 3.6 for how this is done). This assumption is safe, since the specification must already have been analyzed. The task of instantiating the body is left to the Back End; the Back End cannot be run until the body of the generic unit has been analyzed. This procedure has the advantage of allowing the Back End to decide whether to use common code for several instantiations of the same generic unit.

3.3. Name Binding

Each entity of an ADA program is introduced by a declaration with a defining occurrence of the name of that entity. Uses of the entity always *refer back* to this defining occurrence. Attributes at the definition point make it possible for all information about the entity to be determined. The defining nodes for entities together with their attributes play the same role as a *dictionary* or *symbol table* in a conventional compiler strategy. To support the DIANA approach, the appearances of an identifier in the tree have to be divided into defining and used occurrences (see Section 3.1.1).

3.3.1. Defining Occurrences of Identifiers

All declarative nodes (see DECL, Section 2.3.1) have a child which consists of a sequence of one or more nodes representing the identifiers used to name the newly defined entities. These nodes are termed the *defining occurrence* of their respective identifiers; they carry all the information that describes the associated entity. Because the set of attributes which is necessary for this purpose depends heavily on the nature of the denoted entity, we distinguish the defining identifiers according to the nature of the entity which they denote. Thus we have the following set of node types:

```
DEF_ID ::=          argument_id |
                    attr_id |
                    comp_id |
                    const_id |
                    dscrmt_id |
                    entry_id |
                    enum_id |
                    exception_id |
                    function_id |
                    generic_id |
                    in_id |
                    in_out_id |
                    iteration_id |
                    label_id |
                    l_private_type_id |
                    named_stm_id |
                    number_id |
                    out_id |
                    package_id |
                    pragma_id |
                    private_type_id |
                    proc_id |
                    subtype_id |
                    task_body_id |
                    type_id |
                    var_id;
```

The defining occurrence of an enumeration character (DEF_CHAR) and of an operator (DEF_OP) fall into the class of defining occurrences as well.

The consistency of the whole scheme requires that we provide a definition point for predefined identifiers as well. These are pragma names (**pragma_id**), attribute names (**attr_id**), and the names of the arguments of pragmas (**argument_id**). The predefined identifiers are described in Appendix I.

It should be noted that although label names, loop names, and block names in ADA are implicitly declared at the end of the corresponding declarative part, they are not explicitly represented in DIANA. The defining occurrence of a label (**label_id**) is its appearance in a **labeled** statement. The defining occurrence of a **named_stm_id** is its appearance in a **named** statement.

3.3.2. Used Occurrences of Identifiers

All occurrences of identifiers which are not mentioned in Section 3.3.1 are treated as *used occurrences*. The node for a used occurrence of an entity has an attribute (*sm_defn* or *sm_operator*) that refers to the node for the defining occurrence of that identifier (where all information is stored). DIANA distinguishes between three different kinds of usage depending on the context in which the entity is referenced.

```
USED_ID ::=         used_name_id |
                    used_object_id |
                    used_bltn_id;
```

A **used_object_id** is used when the *sm_defn* denotes an object, an enumeration literal, or a number. In all other contexts, the use of an entity is represented by a **used_name_id**, whose only attribute refers to the definition of the entity. Additionally we have a used_char (treated as a **used_object_id**) and a **used_op** (treated as a **used_name_id**). Identifiers for built-in entities are discussed in Section 3.3.4.

3.3.3. Multiple Defining Occurrences of Identifiers

Recall that one of the basic principles of the DIANA design stated that every entity has a single defining occurrence. As this is not the case in ADA itself (*e.g.*, incomplete types, deferred constants), DIANA cannot strictly follow this principle. In the instances where multiple defining occurrences can occur, DIANA uses the following solution. All defining occurrences of an entity that could be multiply defined are represented by a DEF_ID as described above in Section 3.3.1. However, these defining occurrences have an attribute, *sm_first*, that refers to the node for the *first* defining occurrence of the identifier, similar to the *sm_defn* attribute of used occurrences (Section 3.3.2). Nonetheless, the several defining occurrences of the entity all have the same attribute values. The complete details of how DIANA treats multiply defined identifiers are described in Section 3.5.

3.3.4. Subprogram Calls

In ADA it is possible to write built-in operators as function calls and to write user-defined operators as operators. For example,

```
standard."+"(x => 1, y => 2)
```

In DIANA all function calls and operators are represented as function calls. The only exceptions to this method are the short-circuit operators **and then** and **or else** and the membership operators **in** and **not in**, which cannot be overloaded, cannot be represented as functions, and cannot be written as function calls.

DIANA records whether a function call was made using infix or prefix notation through the *lx_prefix* attribute. This information is necessary for subprogram specification conformance rules (Section 6.3.1 of the ADA LRM [8]).

The kind of function call is indicated by the first child of the **function_call** node, which represents the name of the function. This attribute may be a USED_ID or USED_OP, or a selected component where the DESIGNATOR_CHAR child is a USED_ID or USED_OP. This used occurrence

distinguishes built-in operators (or even procedures and entries) from user-defined subprograms.

In a **used_op** or **used_name_id** node, the *sm_defn* attribute denotes the defining occurrence of the user-defined entity. In a **used_bltn_op** (or **used_bltn_id**), the *sm_operator* attribute indicates the built-in entity; this attribute is a private type and is implementation-defined. It represents numeric operators such as "+" and "*", but also represents the implicitly-defined relations for user-defined types.

Derived subprograms are indicated by the original definition from which they are derived. The actual parameters all have type information attached. It is sufficient to compare the actual types to the original ones to determine the implicit type conversion necessary for parameter association if the representation changes. Since type checking has already been performed, if the *sm_exp_type* of an actual parameter is not equal to the *sm_obj_type* of the corresponding formal (in the sense described in Section 3.9), it must be the case that the actual parameter is of a type ultimately derived from that of the formal. Following the chain of derivations starting with the type of the actual parameter will give the sequence of type conversions which must be performed. Similarly for a derived function, the result type of the **function_call** node can be compared with the result type of th function_id.

If a user defines an equality operator for a limited private type, then inequality is introduced implicitly. The user-defined equality is identified by the *sm_defn* attribute of a **used_op** node. In the case of inequality, there is no defining occurrence. The tree is therefore transformed to a standard "not" operation applied to the user-defined equality. This situation is illustrated in Figure 3-2.

The parameter associations for a subprogram call are in the user-written order; it is therefore possible to reconstruct the source program in most cases. It would be awkward to introduce named associations in the case of predefined operators. It would be impossible for implicit ones such as equality, since there is no defining occurrence of the formal parameters. Therefore, DIANA does not normalize parameter associations to named associations. However, DIANA does use the *sm_normalized_param_s* attribute to record the normalized positional list of actuals used in the subprogram call, including any default actual parameters. (The attribute *sm_normalized_comp_s* serves a similar purpose for record aggregates and discriminant constraints).

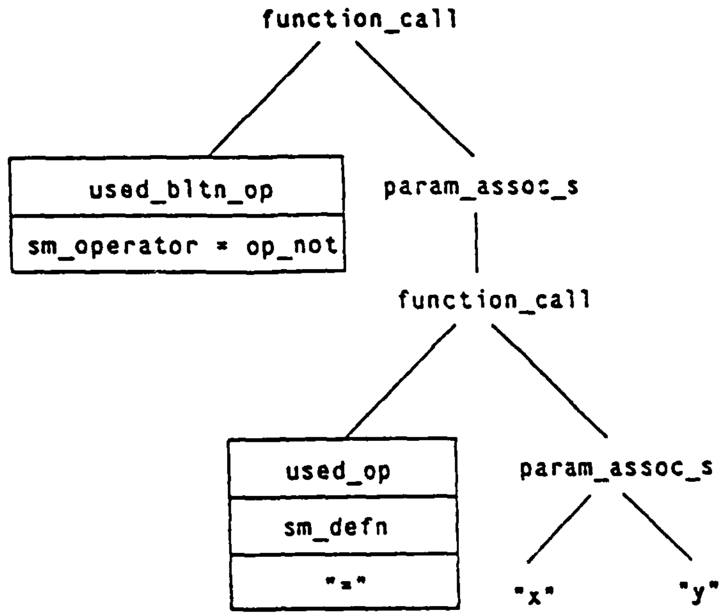


Figure 3-2: Call of Implicitly-Defined Inequality

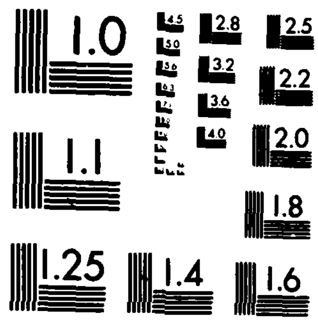
3.4. Treatment of Types

Since anonymous types do not have an explicit declaration in DIANA (see 3.1.3), we cannot use the type identifier as the description of the type. Instead we use the type specification (TYPE_SPEC). In all contexts where structural type information is required, the attributes have values which denote a TYPE_SPEC, *e.g.*, *sm_exp_type* in expressions and *sm_base_type* in **constrained** nodes. This treatment implies that all nodes which can represent a type specification must carry those attributes which describe the detailed type. The meaning of these attributes is explained in the following sections.

It should be noted that most of the attributes described in these sections can be computed from other attributes which are also present in DIANA. The main reason for adding them is that it makes code generation easier. The attributes represent information which the Front End already has and which would be difficult for the code generator to recompute (especially in the presence of separate compilation).

AD-A128 232 DIANA REFERENCE MANUAL REVISION 3(U) TARTAN LABS INC 2/3
PITTSBURGH PA A EVANS ET AL. 28 FEB 83 TL-83-4
MDA903-82-C-0148

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

3.4.1. Machine Dependent Attributes

DIANA originally required machine dependent attributes to be computed because their values were allowed in ADA static expressions and therefore could appear in type declarations. The rules for static expressions (Section 4.9 of the ADA LRM [8]) now only allow attributes of static subtypes in static expressions; attributes whose values are no longer machine dependent.

3.4.2. Type Specifications

There are several ways to specify a type in ADA. Fortunately they all have different syntactic structures so that we are not forced to introduce new node types to carry the different semantic attributes appropriate to each type (as was done for identifiers, see 3.1.1). The following sections give a detailed description of the attributes for each kind of type specification. These descriptions involve the notion of structural type information; this notion is defined in the following section.

3.4.2.1. Structural Type Information

The structural information for a type is expressed by the following nodes of a DIANA Tree:

integer, fixed, float	for numeric types
enum_literal_s	for enumeration types
record	for record types
array	for array types
access	for access types
task_spec	for task types

and the universal types (see Appendix I). Each of these has attributes for values of user defined or implementation chosen attributes.

There are language pragmas (PACK, CONTROLLED) which can be applied to types and which are used instead of a representation specification. Occurrences of these pragmas remain in the DIANA Tree to reconstruct the source, but they are additionally recorded with the type structures they affect using the *sm_packing* and *sm_controlled* attributes.

For record types, there may be representation specifications for the record and its components (including discriminants). A reference to this specification is recorded in semantic attributes of the **record_id**, **comp_id**, and **dscrmt_id** nodes. Similarly for enumeration types, information from representation specifications for the enumeration literals is recorded with the **enum_id**.

3.4.2.2. Subtype Specifications

All subtype indications are represented by a constrained node which has type mark and constraint attributes. The constraint can be void. A subtype declaration can also be used just to rename a type (when no constraint is given), so there may be a sequence of subtype declarations without constraint information. For code generation purposes, it is necessary to know the last applicable constraint, hence a constrained node in DIANA has a corresponding attribute, *sm_constraint*, that points directly to this constraint; the code generator is not forced to walk backwards through the chain of subtype declarations to find the appropriate constraint.

For fixed and floating point types the last applicable constraint may have two parts, a digits (or delta) constraint and a range. In order for the *sm_constraint* to point to the last applicable constraint, a fixed or float node may need to be created for the purpose of representing this constraint. For source reproducibility reasons, the structural constraint may not contain all of the relevant information. Figure 3-3 illustrates the float node that DIANA creates for the following example:

```
type MYFLOAT is digits 6 range -1.0..1.0;
subtype MYFLOAT2 is MYFLOAT digits 2;
```

The code generator also needs the information about the type structure, which is obtained from the original type from which all intermediate derived types and subtypes are constructed. This attribute is named *sm_type_struct*. Note that for derived record and enumeration types it denotes the duplicated type structure, if any. This situation is discussed in the next section, 3.4.2.3.

In a chain of type specifications, a user can add attributes to each type by representation specifications; these specifications are possible only for types, not for subtypes. The type from which a subtype is constructed is called its base type. The attribute *sm_base_type* denotes its type specification, *i.e.*, a derived type (see Section 3.4.2.3) or a type structure (see Section 3.4.2.1) where all representation information can be found. The DIANA structure that results in such a case is illustrated for the following example in Figure 3-4. Note that all information is present at the last subtype declaration: it is an integer type, the values are in the range 1..9, and its representation must not exceed 8 Bits.

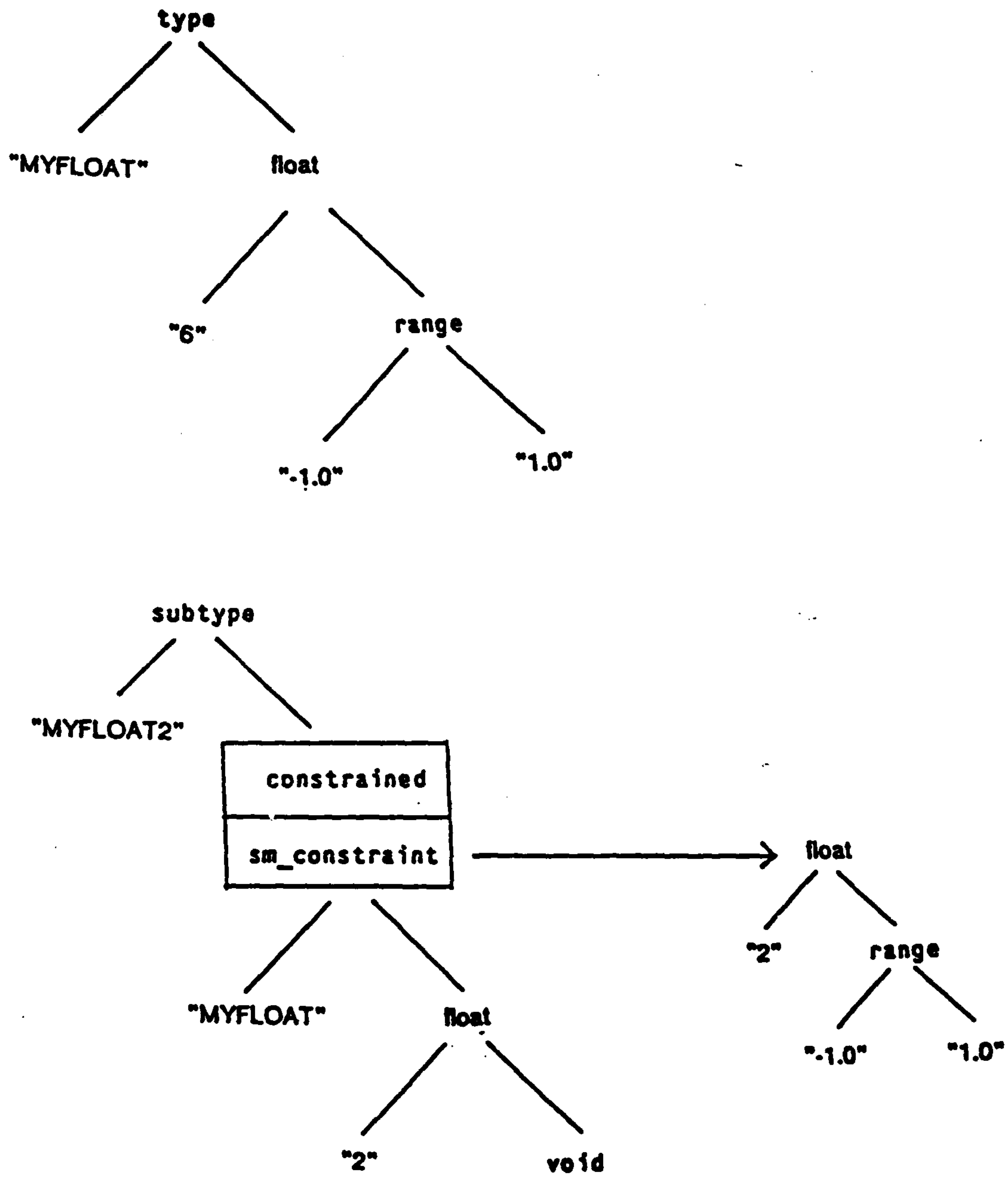


Figure 3-8: Float constraint created by DIANA

```
type T1 is range 1..1000;
subtype T2 is T1 range 1..9;
type T3 is new T2;
subtype T4 is T3;
subtype T5 is T4;
...
for T3'SIZE use 8;
...
```

3.4.2.3. Derived Types

A derived type is used to introduce a new type which inherits characteristics of the parent type. A user can give a new representation specification for every derived type. If no representation is specified, then the attributes of the parent type are inherited. To treat all derived types uniformly, the corresponding DIANA attributes are copied and stored with the derived type specification. The values are overwritten if the user gives a new representation. To support this, the attributes *sm_size*, *sm_storage_size*, *sm_actual_delta*, *sm_packing*, and *sm_controlled*, as well as *cd_impl_size*, are present in a **derived** node.

The subtype indication defines the *parent subtype* and the parent type is the base type of the parent subtype (ADA LRM [8], Section 3.4), so the information about the parent type can be obtained from the subtree of the **derived** node. The corresponding subtype indication is represented by a **constrained** node which has an attribute *sm_base_type* (which denotes the base type) and an attribute *sm_type_struct* (which denotes the structural information for that type); see Section 3.4.2.2.

If this structure is a record or an enumeration type, then it is possible that a representation specification is given for the derived structure—overwriting the old values. For a record structure, these values are recorded with the component declarations (*e.g.*, **comp_id** has the attribute *sm_comp_spec*). In the case of an enumeration type, the values are recorded with the enumeration literal (**enum_id** has an attribute *sm_rep*). The solution of this problem in DIANA requires the creation of a new type structure where the new attribute values can be filled in. This new structure is referenced by the *sm_type_struct* attribute of the **constrained** node of the derived type declaration.

Duplication has another advantage for enumeration literals: since we now have a defining occurrence for a literal, the derivation of an enumeration type introduces new defining occurrences for literals that belong to the derived type and overload the old ones.

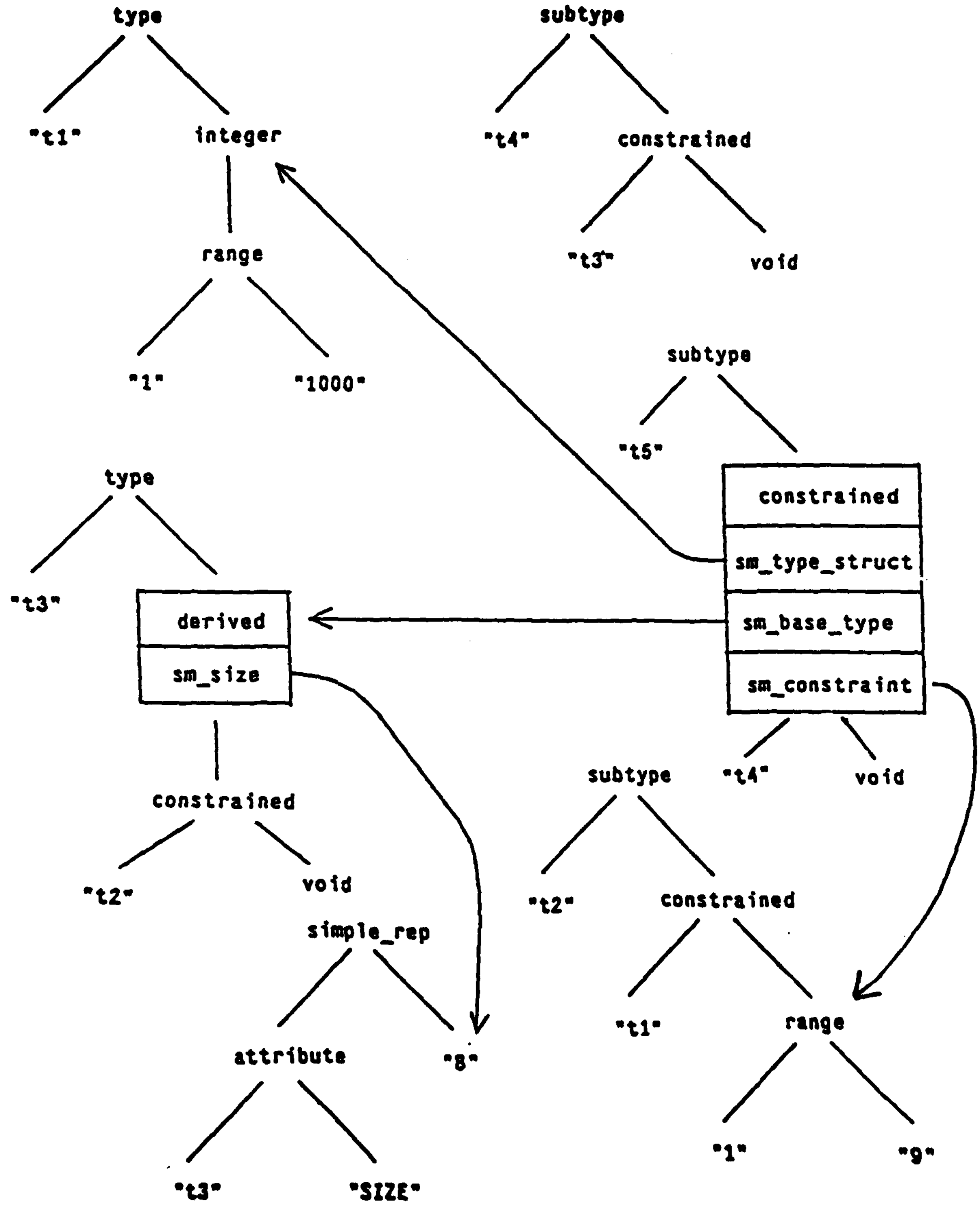


Figure 3-4: DIANA Form of type/subtype Specification

The duplication of the record structure is only meaningful and necessary if a representation specification is given by the user. An implementation of DIANA can choose whether to copy or to denote the old structure. It makes no difference from the logical point of view.

In figure 3-5 we illustrate the DIANA structure that results from the following ADA source.

```
type T1 is (RED, GREEN);
type T2 is new T1;
for T2 use (5, 10);
```

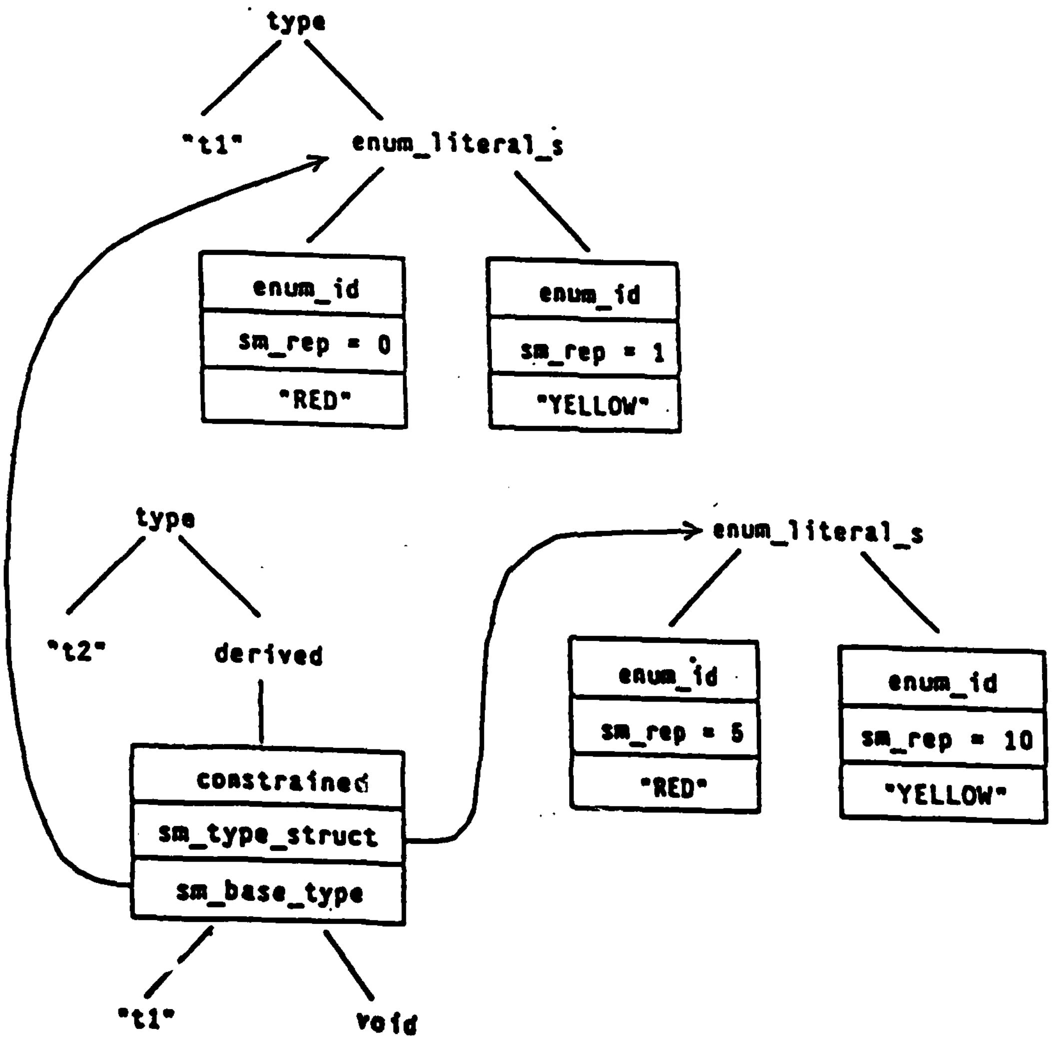


Figure 3-5: An Example for Derived Enumeration Types

3.4.2.4. Incomplete and Private Types

For incomplete and private types, there are two defining occurrences of the same entity. The general solution for entities with several declaration points is discussed in Section 3.5; the approach for incomplete and private types in particular is described in Section 3.5.1.

3.4.2.5. Anonymous Array Types

The ADA rules for multiple elaborations (ADA LRM [8] Section 3.3.1) require that the object declaration:

```
X, Y: array (1..10) of INTEGER := (1..10 => 0)
```

result in X and Y having different types and in fact also cause the aggregate occurring above to be evaluated twice with two different types in the two evaluations. DIANA requires that the var_id's for X and Y refer to different intermediate nodes so that the fact X and Y are different types can be readily determined.

3.4.2.6. Anonymous Derived Types

The ADA semantics require that an integer type declaration is equivalent to a subtype declaration of an anonymously derived integer type (*Section 3.5.5 of the* ADA LRM [8]). To represent this in DIANA without normalizing the source program we have introduced the attribute *sm_base_type* for integer nodes that denotes a derived node that is created to give a unique type definition for the subtype. Similarly, this attribute is also present on float and fixed nodes.

3.4.3. Type Specifications in Expressions

DIANA records the result of overload resolution in every expression node: the *sm_exp_type* attribute denotes the result type of the expression. Additionally, if the value is statically evaluated, the value is recorded in the *sm_value* attribute (see Section 3.8.1).

As far as overloading resolution is concerned, only the base type of an expression is of interest. However, for expressions which denote values which are assured to satisfy a certain constraint, the constraint information is useful. For this reason *sm_exp_type* should refer to a constrained node for (only) the following nodes:

- conversion and qualified whose *as_name* denotes a subtype name.

- indexed and all.

- **function_call**, if the function name is not a built-in operator, and
- **used_object_id** if the object is not declared using an array type specification and is not a single task.

There are three kinds of expressions which implicitly introduce an anonymous subtype: aggregates, slices, and string literals. The resulting subtype can be used to constrain an object if such an expression appears as an initial value for a constant object of an unconstrained array type (ADA LRM [8], Section 3.6.1). The *sm_constraint* attribute is used in these cases to denote a corresponding subtype constraint. Unfortunately, this constraint does not exist in all cases, so it must be computed by creating a suitable structure outside the tree.

In the case of a record aggregate the discriminant values are extracted from the aggregate and used to build a **dscrmt_aggregate** node as a constraint for the type to which the aggregate belongs.

In the case of an array aggregate the constraint attribute denotes a range whose bounds are computed as described in the ADA LRM [8], Section 4.3.2. This range can be used as a constraint for the index type of the underlying array structure.

The *sm_constraint* attribute of a string literal denotes a range whose bounds are computed from the underlying string type (denoted by *sm_exp_type*) and the length of the string literal.

In the preceding two cases, the constraint must be constructed outside the tree. In the case of slices, it is already present; either it denotes the range of the slice itself or, if only a type mark was given, it denotes the range of the corresponding subtype.

Note that because DIANA creates structures outside of the tree, an obvious tree traversal (one that reaches only the structural, '*as_*', attributes) will not yield all of the structural information. Tree traversals that yield all of the structural information do exist; these necessarily follow some semantic attributes as well as the structural attributes.

3.4.3.1. Examples for Constraints of Expressions

Figures 3-6 and 3-7 illustrates the DIANA structure for the following ADA source.

```
type I1 is range 1..1000;
type A is array (I1) of INTEGER;
subtype I2 is I1 range 1..10;
B : A;
```

The figures provide examples for the value of the *sm_constraint* attribute for slices and aggregates.

Figure 3-8 illustrates the DIANA structure for the following ADA source.

```
type MY_STRING is array (INTEGER range <>) of CHARACTER;
C : constant MY_STRING := "ABC";
```

3.4.3.2. Type Specifications for Names

The DIANA class EXP includes the class NAME which can appear in contexts other than expressions (*i.e.*, wherever a name can appear in an ADA program). In all contexts other than expressions, there is no type and no value which can be associated with the nodes representing the name. However, it is not possible to attach different attributes to the same node type depending on the context in which it is used. This section defines the values of these attributes for these cases. (It should be noted that those nodes in the class NAME that can never represent an expression, *e.g.*, any node in the class DEF_ID, do not have the attribute *sm_value*. This discussion is limited to those names that may be used to represent an expression.)

We require that the value of *sm_exp_type* be void for name nodes which are not used to represent expressions. The *sm_value* attribute in these cases must have a distinguished value (see 3.8.1) which indicates that the attribute has not been evaluated. This applies as well to **used_char** when it appears in contexts other than expressions.

Consider the following two ADA fragments.

```
B := P.Q;
I := P.Q'ADDRESS
```

In both cases P.Q is represented by a **selected** node. In the first case it is used in an expression. A type can be attached to the **selected** node, indicating the type of the selected object. In the second case the **selected** node is used to denote an object for which an ADA attribute is to be computed. The node might have a type, as before, but this type is unnecessary since the evaluation of the attribute does not depend on it. A more convincing example is the appearance of a **selected** node in a **with** clause.

Note that the **selected** node does not have a *sm_value* attribute and does not

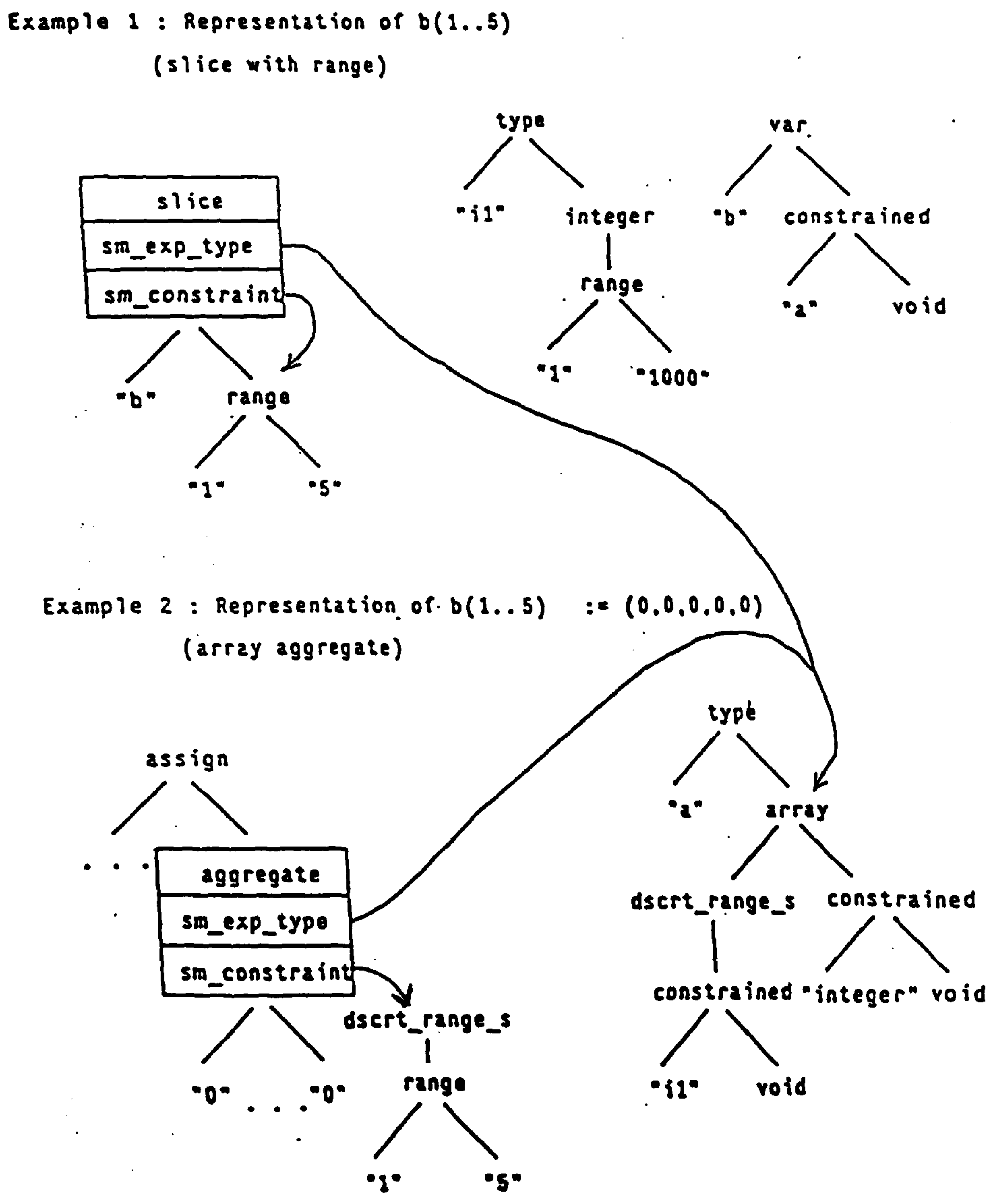


Figure 3-6: Constraints on Slices and Aggregates

Example 3 : Representation of b(i2)

(slice with subtype)

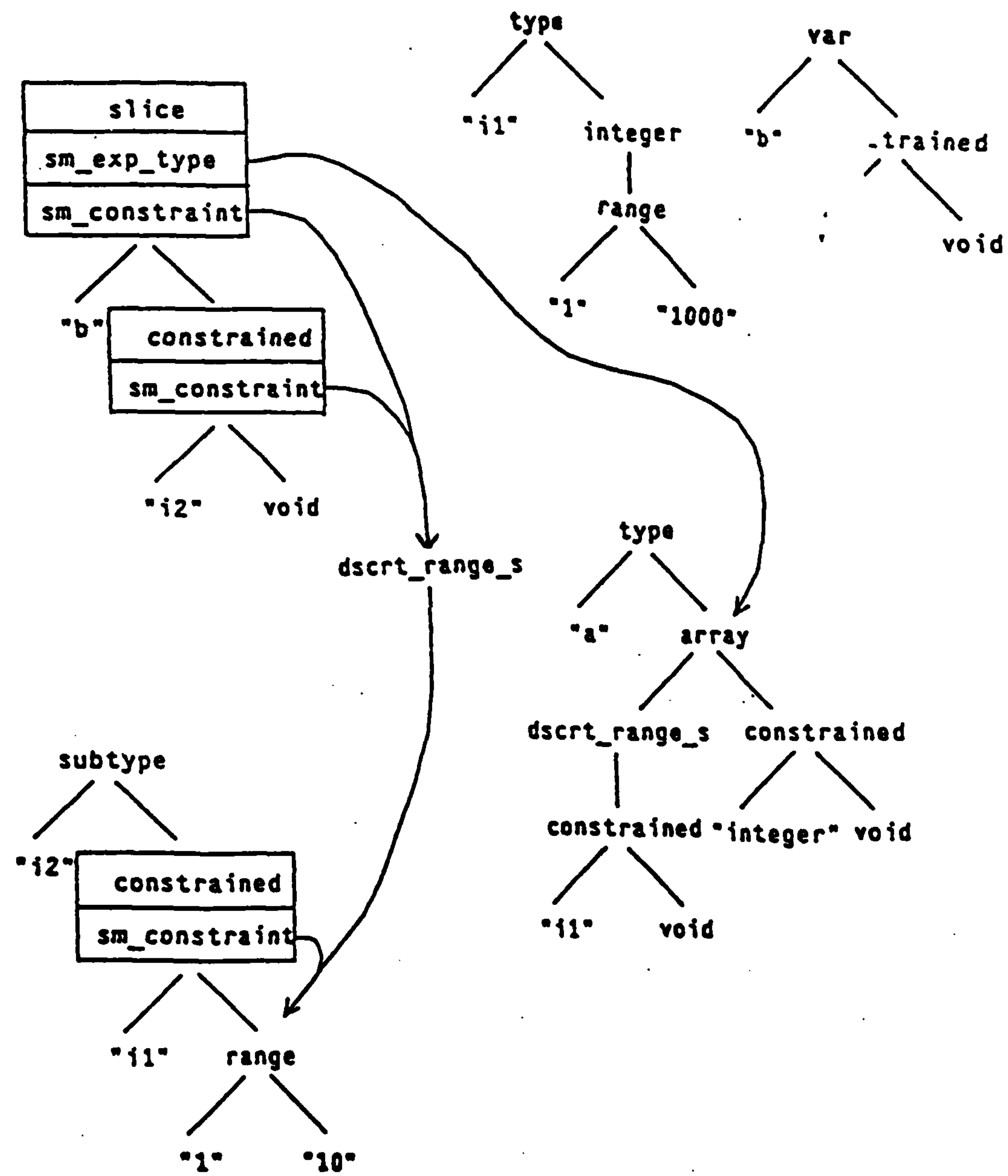


Figure 3-7: Constraints on Slices and Aggregates

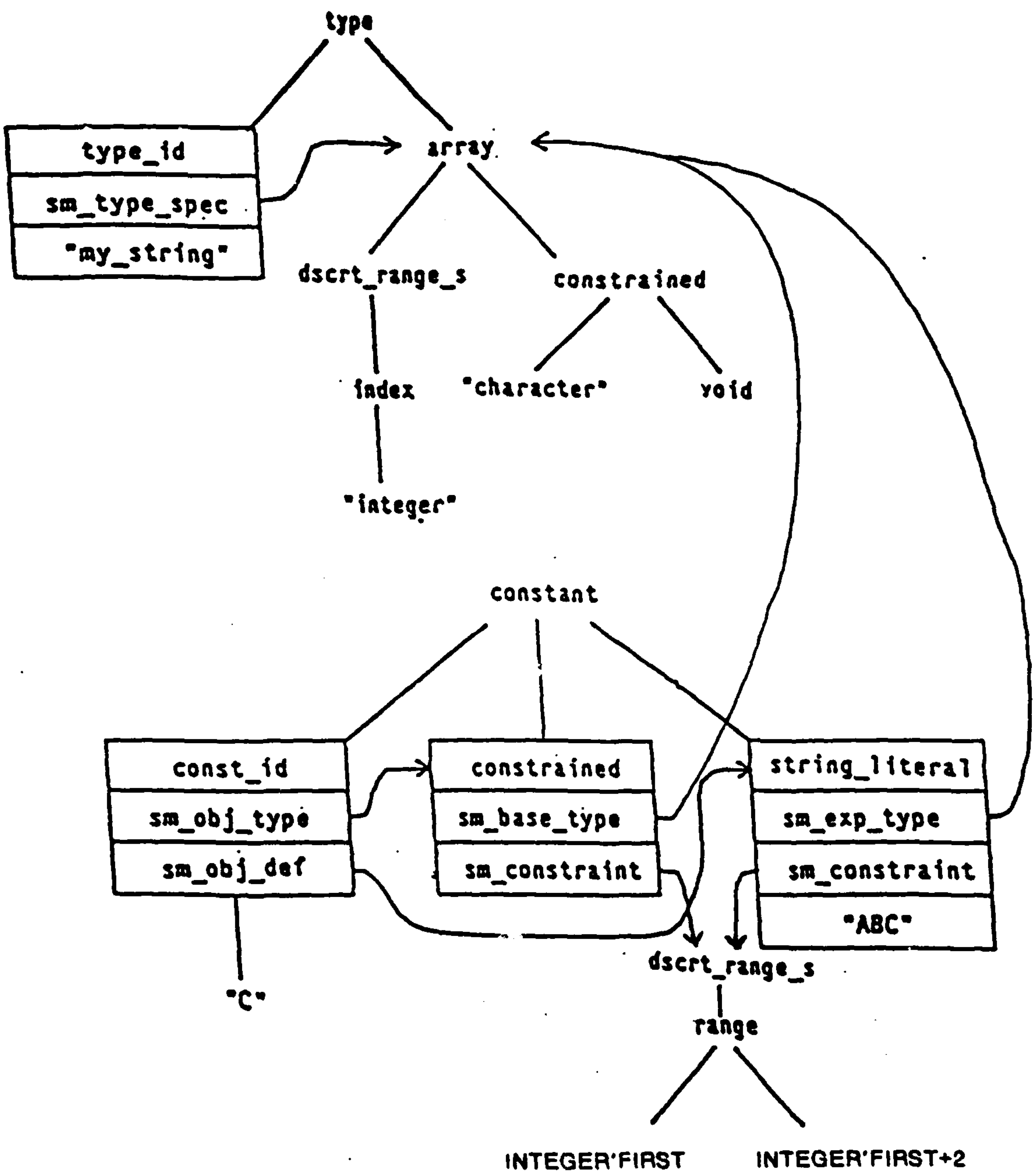


Figure 3-8: Constraints on String Literals

record its value in the context of an expression. Only expressions of *scalar* types can be static (ADA LRM [8] Section 4.9). Thus the DIANA nodes **selected**, **indexed**, **slice**, **all**, and **aggregate** do not have the attribute *sm_value*.

3.5. Entities with Several Declaration Points

One of the basic principles of DIANA requires that there is a single definition of each ADA entity. This conflicts with those ADA facilities that allow or require more than one declaration point for the same entity:

- incomplete type declarations
- (limited) private type declarations
- deferred constants
- subprogram declaration and body
- package declaration and body
- subprogram formals (in the *formal* part of subprogram declaration and body)
- discriminants (in the discriminant part of incomplete or private types)

All instances of multiple defining occurrences are treated as consistently as possible. The principles that apply in all cases are

1. The *first* defining occurrence of an entity is treated as *the* defining occurrence, and

2. all references to the entity should *reference* the first defining occurrence.

All defining occurrences are represented with DEF_ID nodes (Section 3.3.3). Multiple defining occurrences create multiple instances of the same DEF_ID node. DIANA uses the attribute *sm_first* to differentiate among defining occurrences and to allow references back to the first defining occurrence. The attribute *sm_first* references the first defining occurrence of the entity in the same way *sm_defn* denotes the defining occurrence for a **used_id**. The node that is the first defining occurrence has an *sm_first* that references itself.

Note that all used occurrences must reference the same defining occurrence, the one that occurs first. This is the most consistent approach since this is the occurrence that is elaborated in Ada semantics. This requirement allows for a

consistent treatment of all identifiers. The attributes for all defining occurrences must still be determined and for all defining occurrences the attributes must be identical. (The attributes may be different when separate compilation issues intervene: see Section 3.2.1).

There is only one case that deviates from these principles, the case of (limited) private types. Private types are given special treatment in DIANA, as they are in Ada (Section 3.5.1.2).

In the following paragraphs we show the details of the DIANA structure which preserves these principles. We present the details individually for all the cases where the language allows several declaration points of the same entity. (It should be noted that representation specifications are not treated as declaration points, although they do appear in declarative parts.)

3.5.1. Type Declarations

There are two forms of type declaration in which information about the type is given at two different places: private and incomplete types.

3.5.1.1. *Incomplete Type Declarations*

The notion of an incomplete type permits the definition of mutually dependent types. Only the new name is introduced at the point of the incomplete declaration. The structure of the type is given in a second type declaration which must appear in the same declarative part. (This restriction ensures that there is no interference from separate compilation.)

The defining occurrences of both types are described by **type_id** nodes which have the semantic attribute *sm_type_spec*. In both cases, the value of this attribute can denote the full type specification which satisfies the DIANA restriction. The defining nodes also have the attribute *sm_first* which refers to the first occurrence, the incomplete declaration. Note that if the incomplete type declaration includes a discriminant part, that becomes the defining occurrence of the discriminant identifiers (see Section 3.5.1.3 below).

Figure 3-9 illustrates the DIANA structure for the following incomplete type declaration.

```
type T;
...
type T is record ...;
```

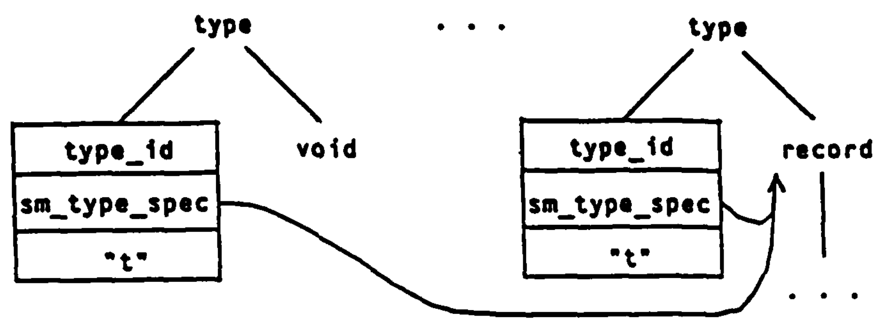


Figure 3-9: Example of an Incomplete Type

3.5.1.2. Private Types

Private types are used to hide information from the user of a package: a private type declaration is given in the visible part of a package without any structural information. The full declaration is given in the private part of the package specification. (This restriction ensures that there is no interference from separate compilation). Unfortunately, we cannot adopt the solution used for incomplete types: if both defining occurrences had the same node type and attributes, we could not determine whether the type is a private one or not. This information is important when the type is used outside of the package.

DIANA views the declarations as though they were declarations of different entities—one is a private type and the other a normal one. Both denote the same type structure in their *sm_type_spec* attribute, however. The distinction is achieved by introducing a new kind of a defining occurrence, namely the **private_type_id**. It has the attribute *sm_type_spec* which denotes the structural information given in the full type declaration. Limited private types are treated in the same way, except that their defining occurrence is a **l_private_type_id**. In the case of (limited) private types the *sm_first* attribute of the **type_id** node refers to the **private_type_id** or **l_private_type_id**.

Figure 3-10 illustrates the DIANA structure for the following example.

```
package DEF_T is
   ...
   type T is private;
   ...
private
   ...
   type T is access ...
   ...
```

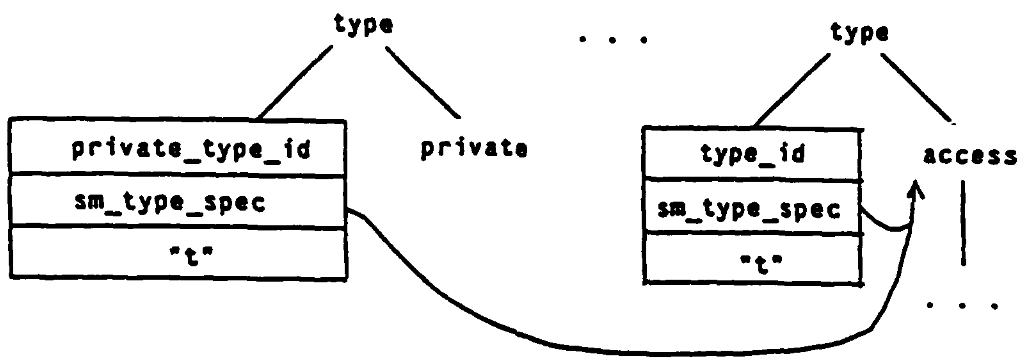


Figure 3-10: **Example of a Private Type**

Since we have introduced two distinct defining occurrences for the private type we must specify which of these definitions a used occurrence refers to. Any use outside of the package denotes the **private_type_id** or **l_private_id** (but nevertheless has structural information) and any usage inside the package denotes the full type declaration; in the interior context, there are no restrictions on the use of the type.

3.5.1.3. Discriminant Parts

When an incomplete type declaration or (limited) private type declaration contains a discriminant part, the discriminant part must also appear in the normal type declaration. This creates a multiple definition of the discriminant identifiers. Thus the **dscrmt_id** node also has an attribute *sm_first* that refers to the first definition point. ADA semantics demand that the discriminant part be elaborated at the first occurrence.

The attribute *sm_discriminants* exists for **l_private** and **private** nodes because for a generic formal private type declaration, the discriminants are not supplied until instantiation. After instantiation, this attribute denotes the discriminants

supplied by the generic actual type.

When a discriminant part *is* supplied in the (limited) private type declaration, the *sm_discriminants* of the **private** node and the **record** node in the normal type declaration should always refer to the discriminants in the first, (limited) private, type declaration.

3.5.2. Deferred Constants

Deferred constants are a direct consequence of the concept of private types; since the structural details of a type are hidden, the structure of the initialization expression must be hidden as well. They are deferred to the private part. The deferred constant declaration (represented by the node **deferred_constant**) and the full declaration of the deferred constant (a **constant** node) are both defining occurrence of the **const_id**. The attributes of both defining occurrences of a deferred constant have the same values, satisfying our requirement. The attributes denote the type specification and the initialization expression. Both attribute values are equal to those of the full declaration of the deferred constant. Note that **const_id** also has the attribute *sm_first* to denote the first defining occurrence. Figure 3-11 illustrates the DIANA structure for the following example.

```
type T is private;
A : constant T;
...
type T is range 0..10;
A : constant T := 0;
```

3.5.3. Subprograms

The declaration and body of a subprogram can be separate from each other. Moreover, in the case in which the body is compiled as a subunit, a stub declaration can also be given. All three declarations can appear in different compilation units: the declaration in a package specification, the stub in the package body, and the body as a compilation unit (subunit) itself. We first examine the simplest case where declaration and body appear in the same declarative part. Then we adapt the solution for the cases where separate compilation is involved.

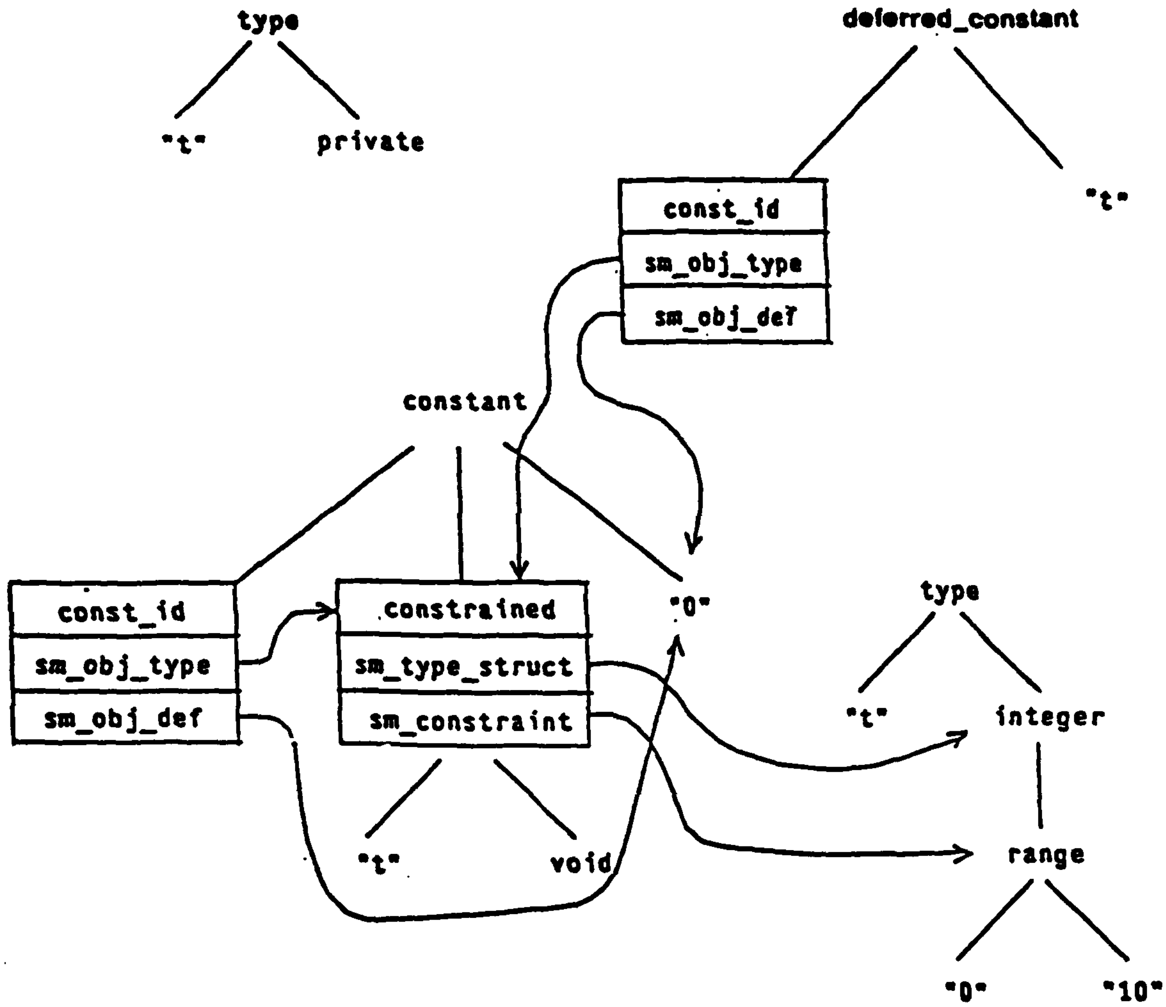


Figure 3-11: Example of a Deferred Constant

3.5.3.1. Declaration and Body in One Declarative Part

The declaration and the body of a subprogram are viewed in DIANA as belonging to the same entity. Therefore, according to our restriction, both defining occurrences must reference the first defining occurrence (the subprogram declaration) and must have the same attribute values. Since the header of the first defining occurrence is used to elaborate the subprogram, the *sm_spec* attribute of both defining occurrences denotes the header of the declaration.

Both defining subprogram identifiers *further reference* the block which describes the body. This method leads to the structure shown in Figure 3-12.

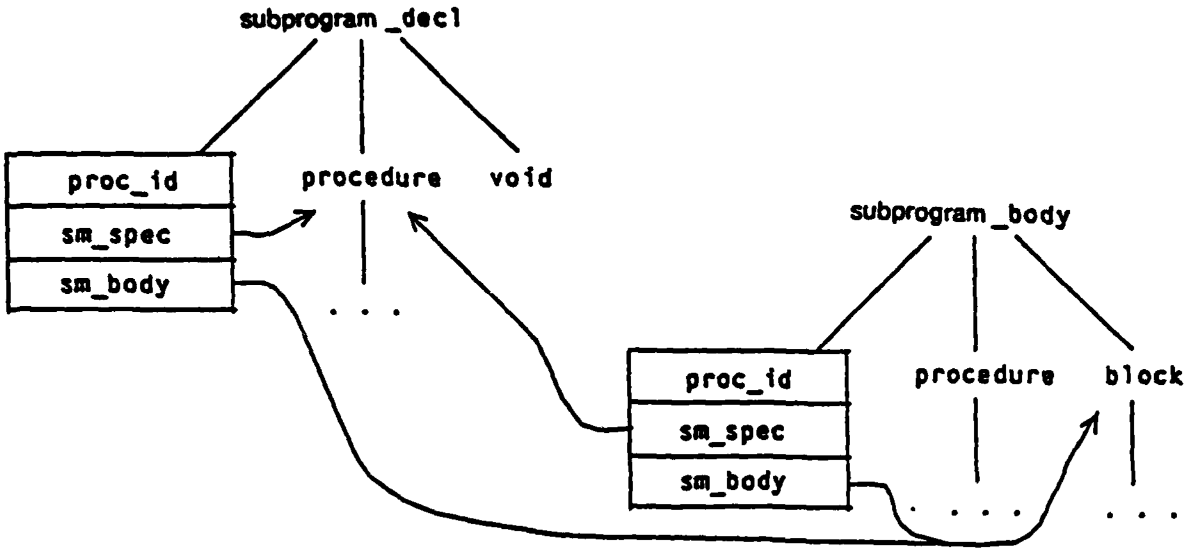


Figure 3-12: Subprogram Structure

3.5.3.2. Declaration and Body in Different Compilation Units

Since a subprogram body cannot appear in a package specification but must be declared in the package body, and since package bodies will often be separately compiled, the declaration and body of the subprogram will often be in separate compilation units.

Updates to previously compiled units are forbidden in DIANA. Therefore, it is not possible to insert the value of *sm_body* in the declaration. The reasons for this decision are discussed in more detail in Section 3.2.1. Therefore, in all cases where the body is in a separate unit, the value of *sm_body* is void. Nevertheless, if the DIANA tree for the declaration is processed, the attribute may be temporarily set to point to the corresponding body if it is present as well. Thus, during processing DIANA principles for multiple definitions are followed.

The permanent structure for a subprogram declaration and body in separate compilation units is as shown in Figure 3-13.

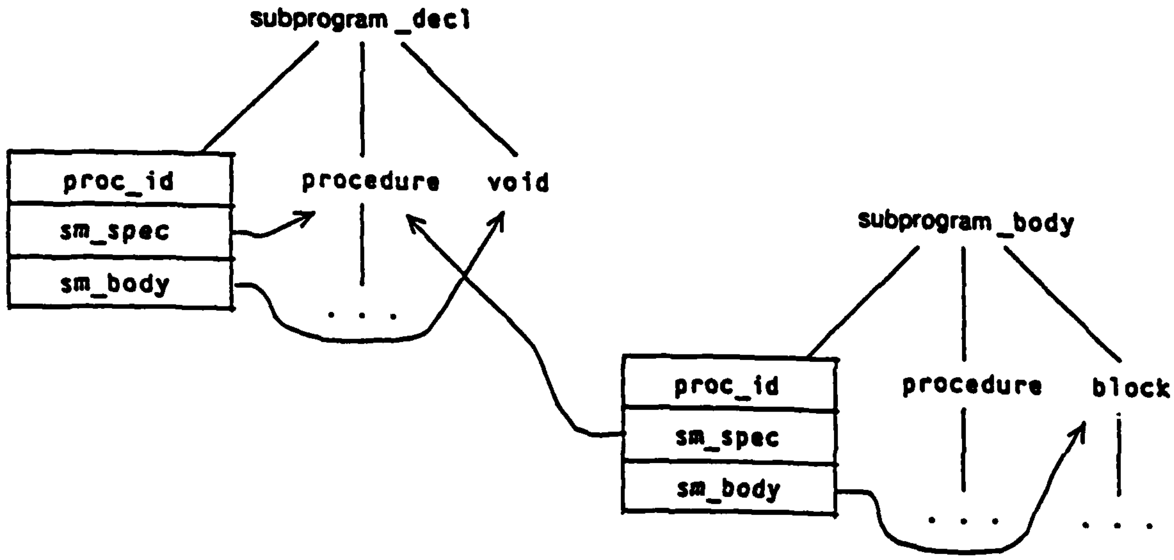


Figure 3-13: Subprogram Declaration and Body in Different Compilation Units

3.5.3.3. Subprogram Bodies as subunits

If a subprogram body is compiled as a subunit, it is possible for there to be a third defining occurrence, a stub declaration, making a defining occurrence in three different compilation units. We adapt the solution presented above, adding the stub declaration which makes the picture more complicated, as is shown in Figure 3-14.

The attribute *sm_stub* is used to refer to the defining occurrence of the stub. This attribute provides a quick means of finding the stub when it is in a separate compilation unit. Figure 3-15 shows the DIANA values for the attributes *sm_first* and *sm_stub*. (In subsequent figures the values for the attributes *sm_first* and *sm_stub* are not shown. The treatment of *sm_first* and *sm_stub* for other DIANA constructs does not differ significantly from the treatment shown in figure 3-15.)

Just as *sm_body* is prevented from forward references, the value of *sm_stub* is required to be void when the stub appears in a separate compilation unit.

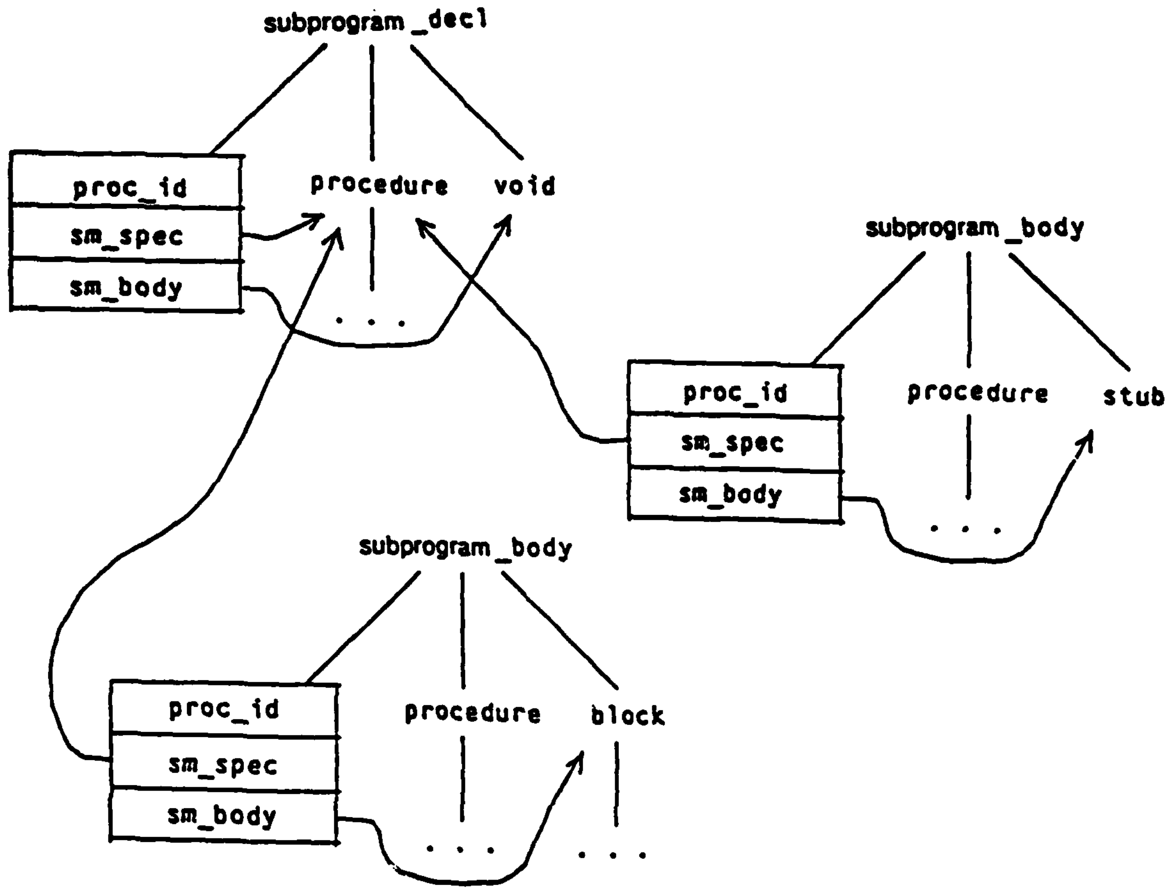


Figure 3-14: Example of a Subprogram Body as a subunit (I)

3.5.3.4. Entries and Accept Statements

An entry declaration and its corresponding accept statements are not treated as different definition points of the same entity. The abstract syntax indicates a name for an accept statement which is viewed as a used occurrence: DIANA uses the same approach. Thus the **entry_id** is the unique defining occurrence: a **used_name_id** appears as a child of an accept statement and refers to the entry declaration. However, the *formal* part of the entry declaration and the accept statement multiply define the entry formals (see Section 3.5.3.5 below).

3.5.3.5. Subprogram Formals

When the declaration of a subprogram is separate from the subprogram body (and stub) the subprogram formal part is repeated. This creates a multiple definition of the subprogram formals. Thus the subprogram defining

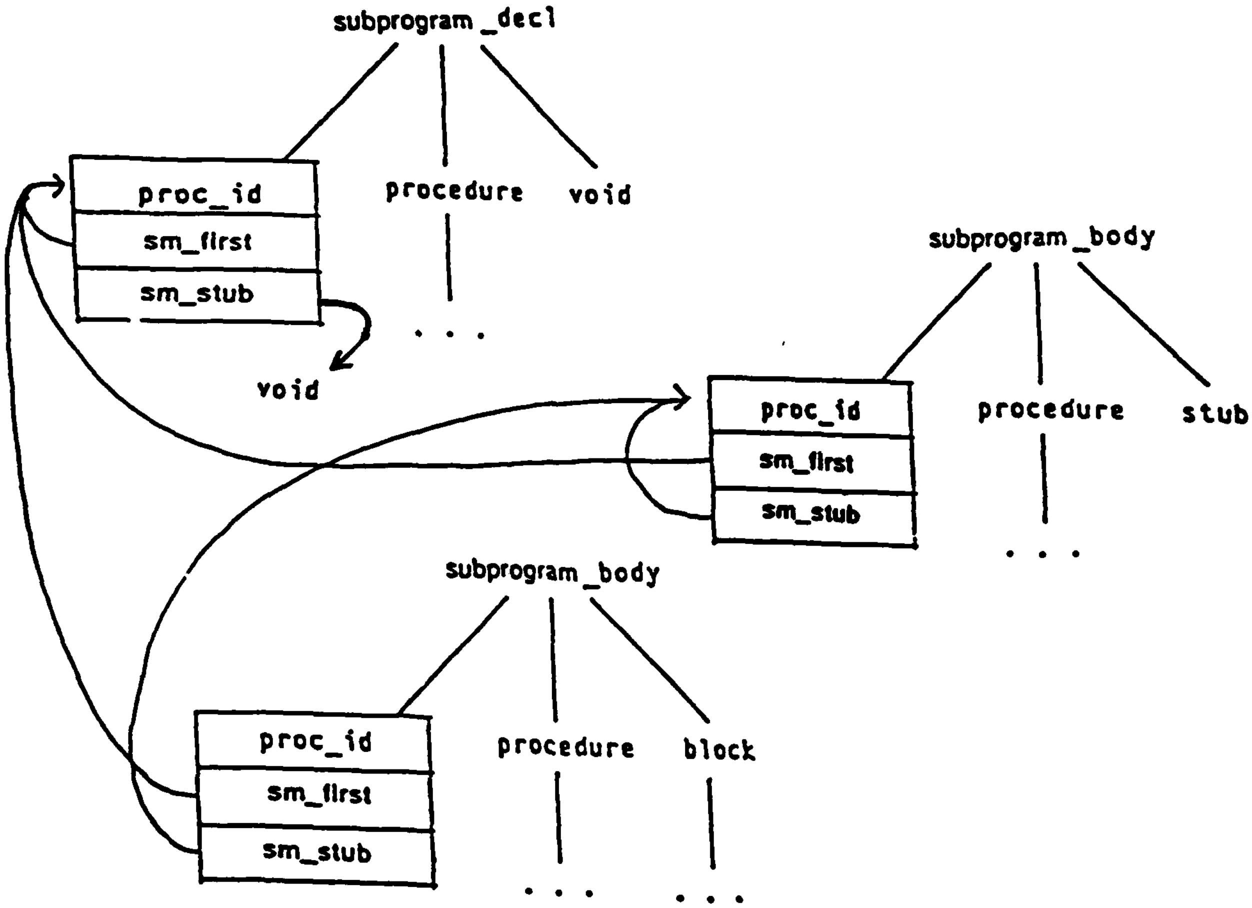


Figure 3-15: Example of a Subprogram Body as a subunit (II)

occurrences (in_id, in_out_id, and out_id) have the attribute *sm_first* to refer to the first occurrence. ADA semantics require that the first occurrence is the one that is elaborated.

This treatment applies to formal parts in entry declarations and accept statements also.

3.5.4. Packages

Packages are declared by at least a specification and possibly a body; in the case of subunits, a stub declaration must also be given. Thus packages present the same situation as subprograms, and the DIANA treatment of packages is in principle the same as that for subprograms (except that the structure and the attribute names are different).

We restrict ourselves to the complicated case of having three different definition places for packages; the DIANA structure is shown in Figure 3-16.

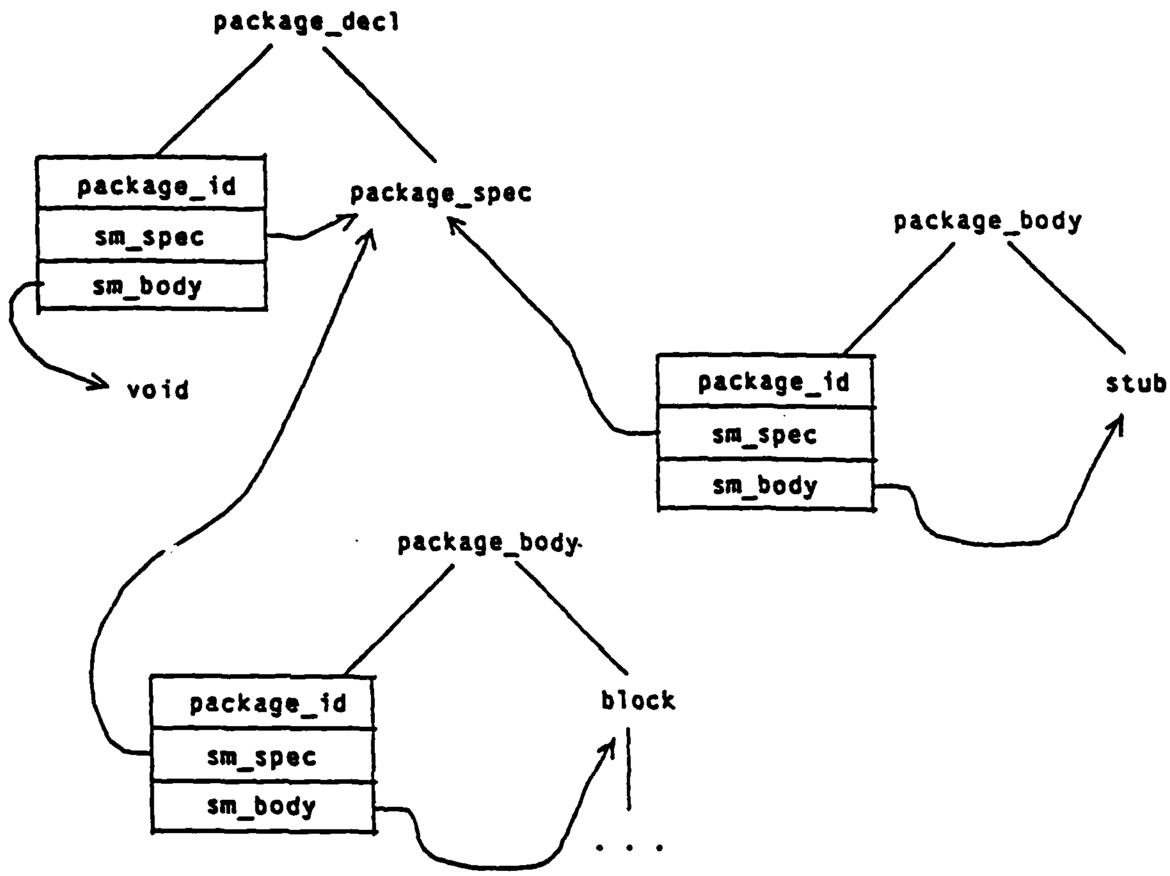


Figure 3-16: Example of a Package Body as a subunit

3.5.5. Tasks

Task specifications can appear in two contexts, as a task type and as a single task specification. The context is distinguished by the kind of the defined identifier (type_id, var_id). A task body is neither, so DIANA has additionally a task_body_id. This additional node implies that there are two defining occurrences and therefore two distinct DIANA entities which do not have the same attributes. Although there are different nodes, the DIANA structure looks similar to the solution for packages. In particular, the same principles are applied in the presence of separate compilation.

3.5.5.1. Task Types and Task Bodies

In the case of a task type and a corresponding body, we have the DIANA structure shown in Figure 3-17. In the presence of separate compilation, the *sm_body* attribute denotes **void** for the task specification and **stub** for the stub declaration. This approach parallels the approach used for for packages and subprograms. Used occurrences of the task identifier denote the **type_id**; the *sm_first* for the **task_body_id** also references the **type_id**.

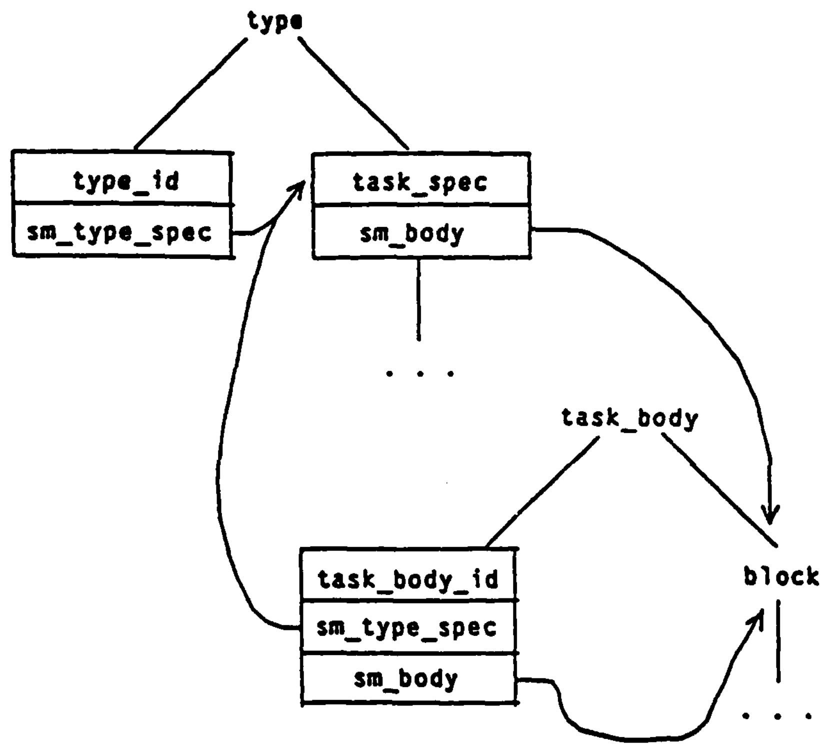


Figure 3-17: Example of a Task Type and Body

3.5.5.2. Single Tasks and Task Bodies

Single tasks are represented by a **task_decl** node with a **var_id**. The task specification is given as an anonymous type specification. The DIANA structure, nearly the same as the structure used for task types, is shown in Figure 3-18. Used occurrences reference the **var_id**; the *sm_first* attribute of the **task_body_id** also references the **var_id**.

Note that in the case of an address specification of a single task, the *sm_address* of the **var_id** and the **task_spec** are both set.

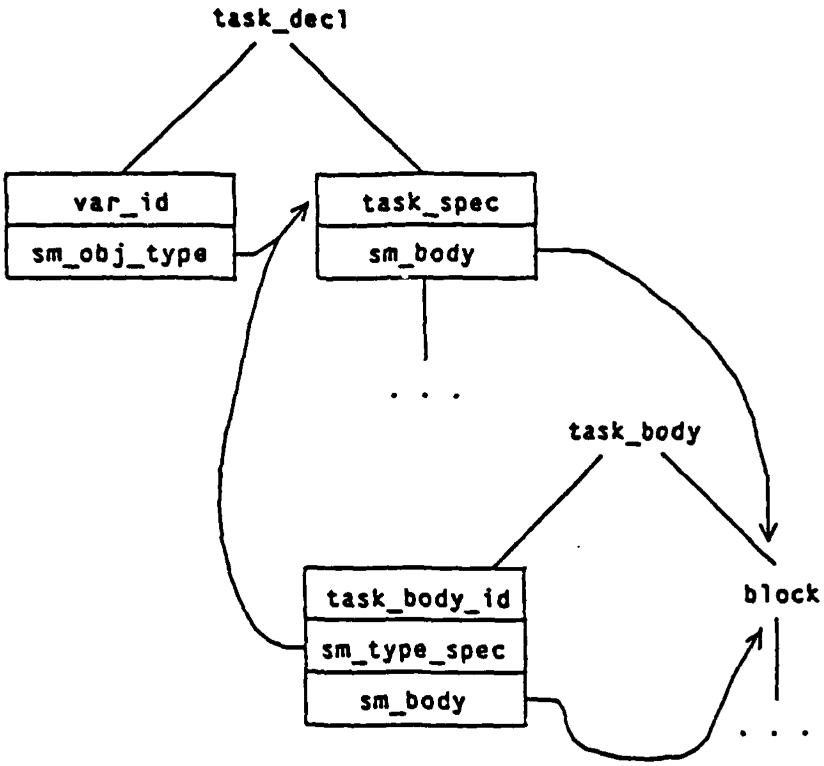


Figure 3-18: Example of Single Tasks

3.5.6. Generic Units

Like subprograms and packages, generic units can have several declaration points: the specification and the body (and possibly the stub as well). In order to have the same information at these declaration points, the identifier of the body of the generic unit has to be a **generic_id** with the same attribute values as the defining occurrence within the specification. Thus the attribute *sm_generic_param_s* points to the list of generic parameters given with the specification, and the attributes *sm_spec* and *sm_body* are set as in the case of simple subprograms or packages. Note that for generic subprograms the subprogram formals are treated as described in Section 3.5.3.5. The DIANA structure for a generic subprogram is illustrated in figure 3-19.

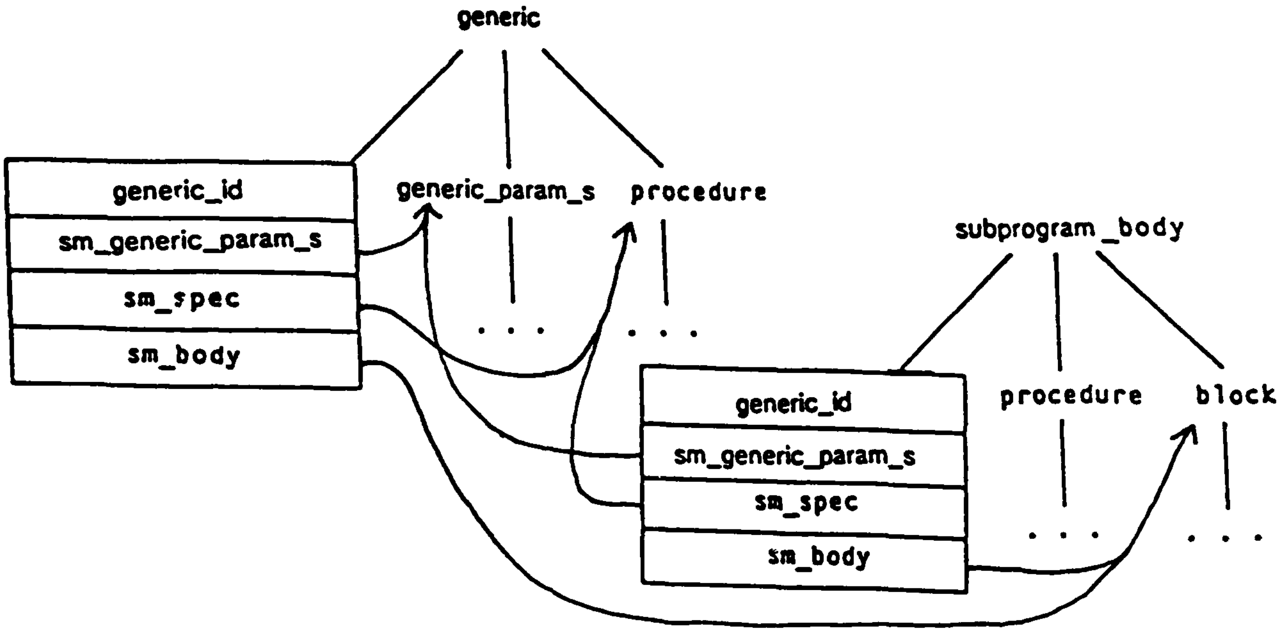


Figure 3-19: Example of a Generic Body as a subunit

3.6. Treatment of Instantiations

In this section we describe how DIANA treats instantiations of generic units.

An obvious implementation would copy the generic unit and substitute the generic actual parameters *for* all uses of the generic *formal* parameters in the body of the unit. This substitution cannot be done if the body of the generic unit is compiled separately. A more sophisticated implementation may try to optimize instantiations by sharing code between several instantiations. Therefore the body of a generic unit is not copied in DIANA in order to avoid constraining an implementation. Indeed, an instantiation may occur in the absence of a generic body.

In DIANA the instantiation is performed in two steps. First, a normalized list of the generic parameters is created. The nodes of the type **instantiation** have the semantic attribute *sm_decl_s* with a sequence of declarations. This attribute is the normalized list of the generic parameters, including entries for all default parameters. The values of this attribute are determined as follows:

- For every generic formal in-parameter, a constant declaration is created (the *sm_obj_def* refers to either the expression given or to its default value).

- for every generic formal in-out-parameter, a variable declaration is created (the *sm_obj_def* refers to a rename node which indicates the object in the actual list that is renamed by the new declaration).

- for every generic formal type, a subtype declaration is created (the *sm_type_spec* attribute is a constrained node with a void constraint that references the type name given in the association list), and

- for every generic formal subprogram, a new subprogram declaration is created (the *sm_body* attribute references a rename node which indicates that the newly created subprogram renames either the subprogram given in the association list or that chosen by the analysis as the default).

In the second step the specification part of the generic unit is copied. Every reference to a formal parameter in the original generic specification is changed to reference the corresponding newly created declaration. If a formal type has discriminants, references to them are changed to point to the corresponding discriminants of the base type of the newly created subtype.

Examples of instantiations are presented in the following two sections.

3.6.1. Instantiation of Subprograms

The generic instantiation of a subprogram is represented by the structure shown in Figure 3-20. We use procedures as an example; the structure for functions is similar. Figure 3-20 illustrates the instantiation of the following generic:

```
generic
LENGTH : INTEGER := 200;                -- default value
type ELEM is private;
procedure EXCHANGE (U : in out ELEM);
...
procedure SWAP is new EXCHANGE(ELEM => INTEGER);
```

The procedure node of the subprogram_decl contains no information; its parameter list is empty. The instantiation node represents the generic parameter associations; it is referenced by the *sm_body* attribute of the proc_id node. The instantiation node also has a normalized list of the generic parameters: it contains a constant declaration of 'LENGTH' using the default and a type declaration of the subtype 'ELEM' using the type name given in the association list. The *sm_spec* attribute of the proc_id node references the header of the instantiated subprogram. It is obtained by copying the generic subprogram's header and replacing references to the generic formal parameters with references to the new subtype declaration and constant declaration.

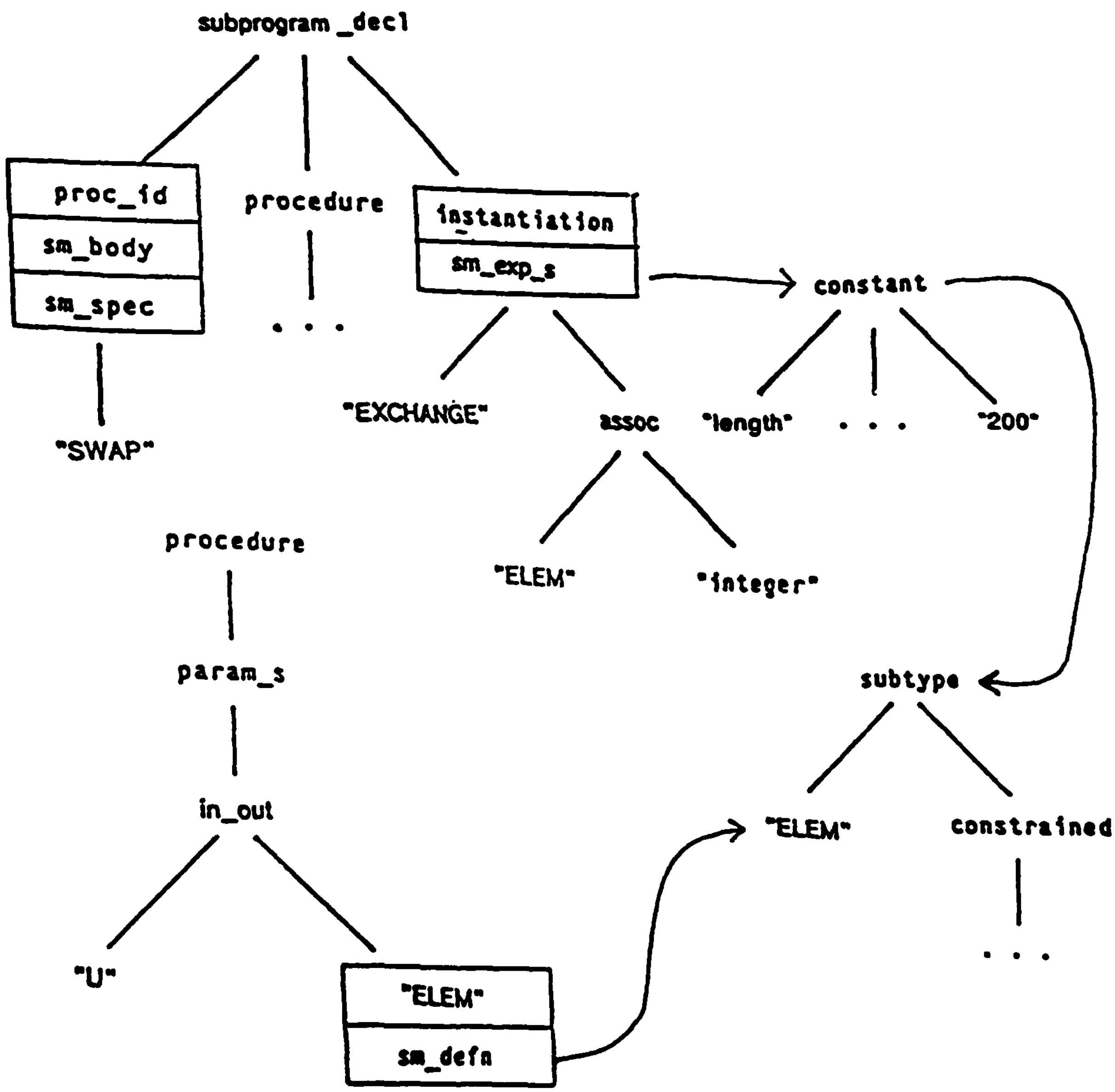


Figure 3-20: Instantiation of a Generic Procedure

3.6.2. Instantiation of Packages

The generic instantiation of a package is represented in DIANA by the structure shown in Figure 3-21. The **instantiation** node is referenced by the *sm_body* attribute of the package identifier. The package specification is constructed by copying the specification of the generic unit and replacing all references to generic formal parameters with references to their corresponding actual parameters. The resulting specification is denoted by the *sm_spec* attribute of the package identifier.

3.7. Treatment of Renaming

The renaming of entities does not introduce further problems. However, the DIANA representation for some renamings may not be obvious. This section clarifies how DIANA treats entities introduced by a renaming declaration.

Renaming of objects and exceptions are simple and not discussed here. Note that an identifier which renames a constant object has to be a **const_id**. Constant objects are constants, discriminants, and parameters of mode in, as well as components of constant arrays.

3.7.1. Renaming of Subprograms

The renaming declaration for a subprogram must repeat the header of the renamed item. This header can be denoted by the *sm_spec* attribute of the newly-introduced subprogram identifier. The rename information is referenced by the *sm_body* attribute, since the actual body can be obtained from the rename information. The structure is illustrated in Figure 3-22.

Note that an identifier which renames an entry or a member of an entry family has to be an **entry_id**. It is possible in ADA to rename an enumeration literal as a function. In such a case the identifier that renames an enumeration literal has to be an **enum_id**.

3.7.2. Renaming of Packages

The renaming declaration of a package does not repeat the package specification. The *sm_spec* attribute of the new package identifier therefore references the original package specification, in order that the specification is always present for a package identifier. The *sm_body* attribute denotes the **rename** node. The resulting structure is illustrated in Figure 3-23.

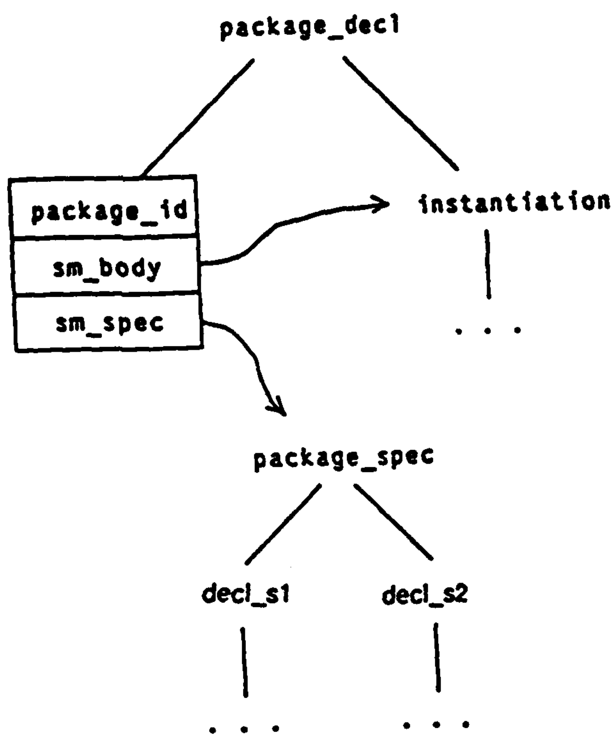


Figure 3-21: Instantiation of a Generic Package

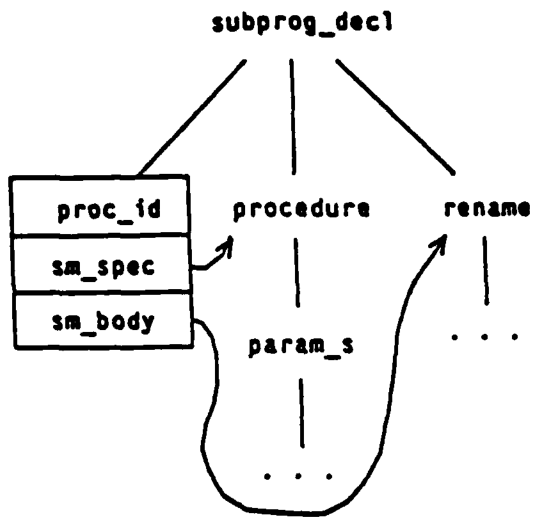


Figure 3-22: Renaming a Procedure

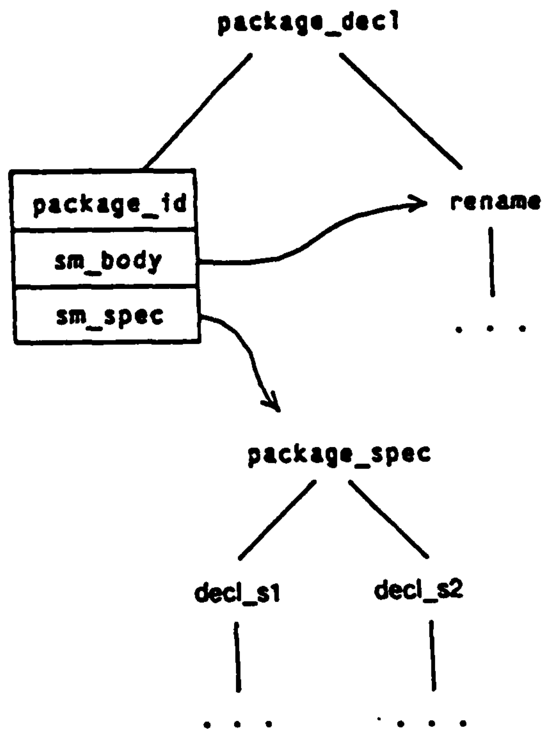


Figure 3-23: Renaming a Package

3.7.3. Renaming of Tasks

Task objects can be renamed like other objects. The task renaming is treated just like the renaming of objects. Task types are renamed just like other types. Note that there is no other renaming declaration for tasks.

3.8. Implementation Dependent Attributes

Representation independence was a principal design goal of DIANA. DIANA does not force an implementation strategy on either a Front or Back End—or on any other tool for that matter. The description of DIANA deals with this problem (in part) by using private types for attributes that are to be implementation defined. An implementation has the freedom to choose a suitable representation, but it must support the corresponding attributes. Thus an implementation must provide appropriate packages in which the attribute types are defined, together with the necessary access operations.

In this section we describe the purpose of the attributes in detail and sketch possible internal and external representations of them.

3.8.1. Evaluation of Static Expressions

The language requires that static expressions be evaluated at compile time in particular contexts (see ADA LRM [8], Section 4.9). This evaluation can be done either by the code generator or by the Front End (with target and host independent arithmetic). Both ways are supported by DIANA. Since the DIANA structure may be used as input to the Front End in the case of separate compilation, the latter solution has the advantage that the previously evaluated expression can be used in the currently compiled unit. For this purpose every expression node that can have a static value has an attribute *sm_value* whose type is implementation dependent[1]. Its external representation is discussed in Chapter 5. The implementation of the type must provide for a distinguished value of this attribute which indicates that the expression is not evaluated. DIANA does not provide for non-static values to be computed, even if an implementation's semantic analyzer is capable of evaluating some such expressions (see Section 1.1.3).

[1]Note that only scalar types can have static expressions

3.8.2. Representation of Identifiers and Numbers

The attribute types *symbol_rep* and *number_rep* are not defined in DIANA. Their external representation is discussed in Chapter 5. Their internal representation is not specified, so that DIANA does not impose a special implementation.

3.8.3. Source Positions

Source position is important for error messages from the compiler. It may also be useful to other tools that work with the DIANA structure, such as interpreters or debuggers.

The structure of this attribute is not defined by DIANA since each computer system has its own notation of a position in a source file. Moreover this notation can vary between tools of the same environment: an interactive syntax-directed editor may have a different type of source position than a batch-oriented compiler for example.

DIANA does not require that this attribute be supported by every implementation (see Section 1.1.3). Any implementation that does support this attribute must define a distinguished value for this type for undefined source positions, which can be used if nodes are created which have no equivalence in the source file.

The library manager for certain implementations may need a value indicating which compilation unit a DIANA entity comes from. This information appropriately belongs with the source position, and should be incorporated into such an implementation's definition of the private type.

3.8.4. Comments

The *lx_comments* attribute is used for recording comments from the source program. The structure of this attribute is not defined by DIANA since every implementation may have its own method of attaching comments to DIANA nodes. A generalized method for attaching comments to nodes is impractical: there is no method that will be accurate for all commenting styles. We envision local commenting standards that will be enforced to match the implementation choices for attaching comments to tree nodes. Note that support of the *lx_comments* attribute is not required for an implementation to be considered a DIANA *producer* or a DIANA *consumer*.

3.8.5. Predefined Operators and Built-in Subprograms

The *sm_operator* attribute is used to identify predefined operators and implementation-dependent built-in subprograms[2]. User-defined operators are treated as functions in DIANA and are not considered here. The predefined operators and built-in subprograms are treated specially because it is important information for the code generator and for an optimizer.

The type of this attribute is implementation-defined. A likely implementation is an enumeration type with at least one literal for each predefined language operator. The refinement of DIANA given in chapter 2 gives the minimum subset of operators that must be supported. An implementation can obviously support further operators which can be added to this enumeration.

The means by which this information is made known to the Front End is not specified in DIANA. We provide *only* for representing the result of semantic analysis: if the Front End recognizes that a compilation unit uses one of the built-in subprograms, then the used_name_id of the subprogram is changed to a used_bltn_id whose *sm_operator* attribute is set to denote the particular built-in subprogram that was used.

3.9. Equality and Assignment

The DIANA representation assumes a well-defined notion of equality for all attribute types, including tree-valued attributes. An implementation must provide an equality comparison operation so that, for instance, the *sm_type_spec* attribute of two entities of the same type will be equal and will not be equal to the *sm_type_spec* attribute of any entity of different type.

If an implementation implements nodes as access types and tree-valued attributes as pointers, then the equality comparison can be a simple pointer equality. DIANA does not force this implementation, however. It is still possible for an implementation to make separate copies of a defining occurrence. For example, consider a situation where a separately compiled unit A defines a type, two other units B and C use this type to declare variables X and Y, and a fourth unit D references both X and Y. It is possible for an implementation to decide to copy some type information from A into B and C. However, a tool processing

[2] For example, an implementation may "build in" knowledge of the LOG function from library package MATH_LIB.

the representation for unit D must be able to compare the *sm_type_spec* attributes of X and Y for equality. Thus the implementation making the copies must keep enough information in its representations to be able to tell that the copies are copies of the same thing. One possible solution is to attach a unique key to every entry and to copy the key along with the other portions of the entity. The equality test can use this key for comparison.

DIANA imposes a further requirement on implementations of attribute-storing procedures. If an implementation stores an attribute of a defining occurrence or a type specification, this change must be visible to all uses of such entities. Once again, making the change visible is easy if the corresponding attributes in the uses are implemented as pointers. In the case where an implementation has copies of such entities, the store procedure must ensure that all copies which might be referenced are updated appropriately.

Note that the duplication of tree structures imposed by DIANA, especially those described in Sections 3.4.2.3 and 3.6, are not copies in the sense of this section. They represent information for new objects, either of derived types or of instantiated units. The new objects must be different from the original ones.

DIANA *does make* a requirement about the value of tree-valued attributes in the external ASCII form (Chapter 5). Tree-valued attributes that are equal must be represented externally by a reference to the same tree; they must essentially share the value. This issue is addressed more completely in Chapter 5.

3.10. Summary of Attributes

A short description of all attributes of DIANA closes the Rationale. We do not describe the structural attributes (for the tree); this description is in the AFD and can be deduced from the concrete syntax of ADA (which is included in the DIANA *definition for convenience*). The remaining three attribute classes are described. If they are already explained in the Rationale, then only a reference to that section appears.

3.10.1. Lexical Attributes

lx_numrep: internal (or external) representation of a numeric literal, the type is implementation dependent, see 3.8.2.

lx_default: is of type Boolean, indicates whether the mode of an in-parameter was specified (False) or defaulted (True).

lx_prefix: is of type Boolean, indicates whether a function call was written using prefix (True) or infix (False) notation, see 3.3.4.

lx_srcpos: source position of the corresponding node, the type is implementation dependent, see 3.8.3.

lx_symrep: internal (or external) representation of a symbol (*i.e.*, an identifier or a string), the type is implementation dependent, see 3.8.2.

lx_comments: representation of comments from the program source, the type is implementation dependent, see 3.8.4.

3.10.2. Semantic Attributes

sm_actual_delta: is of universal rational number type, contains the value of the predefined attribute 'ACTUAL_DELTA.

sm_address: denotes the expression given in a representation specification for the predefined attribute 'ADDRESS. It is void if the user has not given such a specification.

sm_base_type: denotes the base type of a subtype, see 3.4.2.2.

sm_bits: is of a universal integer type, contains the value of the predefined attribute 'BITS.

sm_body: denotes the body of a subprogram or package. It is void if the body or stub are not in the same compilation unit, see 3.2.1. For instantiated or renamed entities it has the type instantiation or rename (see 3.6 or 3.7, respectively). For generic formal subprograms it denotes the FORMAL_SUBPROG_DEF. If the pragma INTERFACE has been applied to the subprogram, it denotes the defining occurrence of the given language name in the predefined environment(see Appendix I).

sm_comp_spec: refers to the representation specification for a record component or discriminant.

sm_constraint: for expressions see 3.4.3, for subtypes see 3.4.2.2.

sm_controlled: indicates whether the pragma CONTROLLED has been applied to the type.

sm_decl_s: belongs to an instantiation node. It refers to a normalized parameter list which contains a declaration (DECL) node for all formal parameters, see 3.6.

sm_defn: denotes the defining occurrence of a used identifier, see 3.3.

sm_discriminants: denotes the sequence of discriminants given for a record or (limited) private type, may be empty, see 3.5.1.3.

sm_exception_def: denotes the EXCEPTION_DEF subtree of an exception declaration, which is void in normal cases and a rename node if it is a renaming declaration.

sm_exp_type: denotes the type of the expression as the result of overloading resolution, see 3.4.3.

sm_first: refers to the first occurrence of a multiply defined identifier, see 3.3.3.

sm_generic_param_s:
denotes the list of generic parameters of a generic subprogram or package.

sm_init_exp: denotes the initialization expression given for numbers, in parameters, record components, and discriminants.

sm_location: denotes the location of a subprogram: it may be (a) void, (b) the identifier (pragma_id) of the pragma INLINE if that has been applied to the subprogram, or (c) an expression supplied by the user in an address specification for the subprogram.

sm_normalized_comp_s:
denotes the normalized list of values for a record aggregate or for a discriminant constraint, including default values.

sm_normalized_param_s:
denotes the normalized list of parameters for a procedure, function, or entry call, including the default parameters.

sm_obj_def: denotes the initialization expression of an object. It is void if none is given. In the case of a renamed object, it denotes the rename node of the declaration structure.

sm_obj_type: denotes the type specification of a declaration (constants, parameters, discriminants, numbers, variables, enumeration literals, and tasks). For deferred constants see 3.5.5. In case of numbers it denotes one of the universal types, see Appendix I.

sm_operator: denotes one of the predefined operators or built-in subprograms, see 3.8.5.

sm_packing: indicates whether the pragma PACK has been applied to that type.

sm_pos: is of universal integer type, contains the value of the predefined language attribute 'POS of an enumeration literal.

sm_record_spec: refers to the representation specification for a record.

sm_rep: is of universal integer type, contains the value of the predefined language attribute 'VAL of an enumeration literal, which can set by the user. See also 3.4.2.3.

sm_size: denotes the expression given in a representation specification for the predefined language attribute 'SIZE; it is void if the user has not given such a specification.

sm_spec: denotes the *specification of a subprogram* or package. In the case of subprograms, it is its header (for instantiations, see 3.6). In the case of packages, it is the package specification. For instantiated packages, see Section 3.6 and for renamed packages, see Section 3.7. In the case of a generic unit, it is the generic header of the unit.

sm_stm: denotes the statement to which a label, loop name, or block name definition belongs or the loop which is left by an exit statement.

sm_storage_size: denotes the expression given in a representation specification for the predefined language attribute 'STORAGE_SIZE; it is void if the user has not given such a specification.

sm_stub: refers to the defining occurrence of the stub, see 3.5.3.3.

sm_type_spec: denotes the specification which belongs to a type identifier; for private and incomplete types, see Section 3.5.1, for tasks and task body identifier, see Section 3.5.5.

sm_type_struct: denotes the structural information of a subtype, see 3.4.2.2, or derived type, see 3.4.2.3.

sm_value: contains the value of the corresponding expression if it is statically evaluated. Its type is implementation dependent, see 3.8.1.

3.10.3. Code Attributes

cd_impl_size: of type universal integer, contains the value of the attribute 'SIZE for static subtypes. It may be less than a user defined size.

3.10.4. Unnecessary Attributes

There are a number of attributes one might expect of semantic analysis that are not explicitly represented in DIANA since they are very easy to recompute.

The floating point attributes corresponding to 'MANTISSA, 'EMAX, 'SMALL, 'LARGE, and 'EPSILON can all be computed from 'DIGITS, which is required to be a static expression. Formulae for these attributes are given in Sections 3.5.7 and 3.5.8 of the Language Reference Manual, and are reproduced here for convenience:

```
'MANTISSA     = ceiling('DIGITS * Ln(10) / Ln(2))

'EMAX         = 'MANTISSA * 4

'SMALL        = .5 * 2**(-'EMAX)

'LARGE        = (1.0 - 'EPSILON) * 2**'EMAX

'EPSILON      = 2.0**(-'MANTISSA)
```

For fixed point types, all attributes can be defined in terms of 'ACTUAL_DELTA and 'BITS.

CHAPTER 4
DEFINITION OF THE DIANA OPERATIONS

Recall that DIANA is an abstract data type. By the nature of an abstract data type as implemented in a programming language, all that need be known about the type are the functions and procedures that operate on objects of the type. Thus to realize the abstract type DIANA in some programming language, all that is needed is to write those functions and procedures. In a language like ADA it is possible to separate the *specification* of these functions and procedures from their *implementation*.

In this chapter we provide an ADA specification (but not implementation) of the interface to the necessary functions and procedures to define DIANA. Further, we suggest how, in general, an implementation-specific package may be derived from an IDL definition. Since the derivation of packages from an IDL description is a complex topic, we only sketch one possible derivation for one particular language. A detailed discussion of the package derivation process is given in the IDL Formal Description [9].

4.1. The DIANA Operations

Every object of type DIANA is the representation of some specific ADA program (or portion of an ADA program). Specifically, it may be thought of as the output from passing that program through the Front End of an ADA compiler. A minimum set of operations on the DIANA type must include the following functions and procedures:

type_getter — Such a function permits the user to determine of a given object what its type is. In DIANA terms, if an object is known to belong to some specific node class, the function determines the object's node type.

selector — Such a function returns the value of a specific attribute of a node.

constructor — Such a procedure builds a node from its constituent parts, or changes the value of an attribute of a node.

In addition, operators are necessary to determine the *equality* of DIANA objects. Specifically, are a given pair of instances of a DIANA type in fact the *same*

instance, as opposed to *equivalent* ones[1]? In case there are variables of this abstract data type, an *assignment* operator is necessary as well.

4.2. DIANA's Use of Other Abstract Data Types

An IDL definition (such as the definition of DIANA in Chapter 2) is built upon subsidiary abstract data types. These include those used in the IDL notation (such as `Integer`, `Boolean`, `Seq Of`) as well as implementation-defined attribute types (such as source_position, symbol_rep, and so on). All of these except `Seq Of` have the same operations as described above. It must be carefully noted that for the scalar types (`Integer`, `Boolean`) there is usually no distinction drawn between equality and equivalence. Whenever doing so is necessary, we carefully draw such a distinction.

The sequence type `Seq Of` can be considered as a built-in type that has a few special operators. Specifically, there must be a way to check if a sequence is empty and to fetch items from a sequence. Additionally, there must be operators for adding and removing items from a sequence.

The implementation defined types must have all the operations appropriate to them as well as those described above for attributes and nodes.

4.3. Summary of Operators

This section summarizes the operations described above.

The operations on *nodes* are

- create a node;
- fetch the value of an attribute of a given node;
- set the value of an attribute of a given node;
- compare two nodes to see if they are the same node; and
- assign a specific node to a variable.

The operators defined for the IDL sequence type (an ordered list of nodes of the same class) are

- create a sequence of a given type;
- select an element of a sequence;
- add an element to a sequence;

---

[1] This distinction is addressed further in Section 3.9 on page 123.

- remove an element from a sequence;
- compare two sequences to see if they are the same sequence; and
- assign a sequence to a variable of sequence type.

The operators required for the IDL scalar types (Integer, Rational, and Boolean) are

- create a scalar;
- compare two scalars to see if they are equal (*i.e.*, the same scalar); and
- assignment.

4.4. General Method for Deriving a Package Specification for DIANA

To derive a general package specification for defining this abstract data type called DIANA, further decisions concerning the implementation model need to be made. For example, one must decide how to represent the various DIANA objects. After these decisions have been made, a straightforward process can be applied to derive the package specification from the DIANA domain. A formal method for specifying these decisions is presented in the IDL formal description. Indeed, an IDL tool would produce such a package automatically from the *definition of* DIANA *in Chapter 2*. For the purposes of this document, the following discussion is sufficient.

The *implementation model* must deal with two separate areas of concern. First, there are the implementation restrictions imposed by the choice of the source language that the DIANA type is being implemented in. Secondly, there is the choice of corresponding entities in the implementation language for entities in the DIANA domain (*i.e.*, how DIANA objects are represented). These decisions can be driven by the design considerations of tools that expect to use the DIANA type, as well as by specific restrictions of the host system.

The general steps are as follows:

- *representation of IDL types*. An implementation for each of the IDL types must be chosen. Normally for the scalar types, the implementation language supports an equivalent (or close enough) abstract type. For the sequence type and the implementation defined types, the same decisions need to be made, and an abstract data type for these derived and specified. (The DIANA domain specification provides a handle on the abstract data types for the implementation defined types.)

- *representation of node classes*. The class names of the DIANA language must be handled by the package derivation process, because

the types of the attributes are defined using these meta-variables.

- *representation of nodes*. The node representation choice must permit attribute values to be associated with the node, since each specific instance of a node may have different attribute values.

- *method of defining operators*. The operators in the language must be specified either as functions and procedures in the implementation language or by equating them to specific operations already in the implementation language.

4.5. Deriving a Specific ADA Package

To derive a specific ADA package, we apply the general method as outlined in the previous section. First, we choose an implementation model of an abstract data type defined as a single package. A single ADA **private** type is used to define all nodes in the DIANA domain. All operations are calls on procedures or functions specified in the package. Having made these decisions, we then address the following points:

- *representation of IDL types*. The IDL `Boolean` type could be implemented directly by the ADA BOOLEAN predefined type. However, the IDL `Integer` and `Rational` types would have to be represented somehow so as to be able to represent arbitrarily large quantities, and (in the case of rationals) to represent them exactly with no approximation[2]. Using the ADA predefined types INTEGER and FLOAT would not be adequate.

  For the sequence type `Seq Of`, we include a private type definition and primitive operations. The operations permit creation of an empty sequence (**Make**), functions to add an element at the beginning (**Insert**) and end (**Append**) of a sequence, and functions for selecting the first element of a sequence (**Head**) and the remainder of a sequence (**Tail**). There is also a function to determine if a list is empty (**Is_Empty**). Note that additional functions and procedures for this type could be added.

- *representation of node classes*. Since a single type is being used to represent all nodes in the domain, the distinction between different classes is not necessary.

- *representation of nodes*. A single private type (called **Tree**) is provided for all the node names defined in the DIANA domain. An enumerated type (called **Node_Name**) is defined which provides a

[2] These requirements are spelled out in the Ada LRM, which requires that some arithmetic performed at compile time be done exactly.

name for all the various nodes defined in the DIANA domain. An additional function (named **Kind** and returning a result of type **Node_Name**) is added to the **Tree** type to distinguish between different node kinds.

- *method of defining operators*. The create operator for the various nodes becomes a single function that takes a **Node_Name** and returns a new **Tree** node with most of its attributes not defined. Each of the DIANA attributes has a corresponding procedure and function in the package specification that respectively modify and fetch the value of an attribute. The procedure and function both take the specific **Tree** node as an argument. The procedure takes an additional argument which gives the new value for the attribute; the function returns the corresponding attribute value.

The *comparison* operators for the nodes and for sequences are the built-in ADA comparison operators ('=', '/=') which are defined for private types. The comparison operators for the scalar types are not defined in this package. The ADA language provides all *create* operations for the scalar types. The *assignment* operators are the pre-defined ADA assignment operators for variables of the private types. Except for these assumptions on the use of built-in operations, the full ADA package is given.

A few facts are important:

- Because some of the DIANA node types conflict with ADA reserved words, we choose to prefix all **node_names** with the prefix "dn_" (short for DIANA).

- Remember that this specification defines a minimal set of operations; implementations may augment it with other useful ones for particular applications.

- We have added an additional type (ARITIES) and several procedures and functions (ARITY, Son1, Son2, and Son3) which are mentioned in the ADA Formal Definition and which are very useful in the tree traversals essential to many phases of compilers, as well as other tools.

4.6. The DIANA Package in ADA

A summary of essential points of the ADA package specification for DIANA appears in Figure 4-1 on page 134. For ease of understanding, the figure contains only as much of the package as fits onto one page.

The package defines and makes available the following types, functions, and procedures:

```
package Diana is
    type Tree is private;                    -- a Diana node
    type SEQ_TYPE is private;                -- sequence of nodes

    type NODE_NAME is                        -- enumeration class for node names
     (  ...                                  -- about 160 different node types
      );

    -- Tree constructors.
    function  MAKE     (c: in NODE_NAME)       return TREE;
    procedure DESTROY  (t: in TREE);

    function  KIND     (t: in TREE)            return NODE_NAME;

    -- Tree traversers from the Ada Formal Definition.

    type ARITIES is (nullary, unary, binary, ternary, arbitrary);

    function  ARITY    (t: in TREE)            return ARITIES;
    function  SON1     (t: in TREE)            return TREE;
    procedure SON1     (t: in out TREE; v: in TREE);
    function  SON2     (t: in TREE)            return TREE;
    procedure SON2     (t: in out TREE; v: in TREE);
    function  SON3     (t: in TREE)            return TREE;
    procedure SON3     (t: in out TREE; v: in TREE);

    -- Handling of list constructs.
    function  HEAD   (l: in SEQ_TYPE)        return TREE;        -- LISP CAR
    function  TAIL   (l: in SEQ_TYPE)        return SEQ_TYPE;    -- LISP CDR
    function  MAKE                           return SEQ_TYPE;
                                                   -- return empty list
    function  IS_EMPTY (l: in SEQ_TYPE)      return BOOLEAN;
    function  INSERT (l: in out SEQ_TYPE;
                      i: in TREE)            return SEQ_TYPE;
                                                -- inserts i at start of l
    function  APPEND (l: in out SEQ_TYPE;
                      i: in TREE)            return SEQ_TYPE;
                                             -- inserts i at end of l

    -- Handling of LIST attribute of list constructs.
    procedure LIST     (t: in out TREE; v: in SEQ_TYPE);
    function  LIST     (t: in TREE)          return SEQ_TYPE;

    -- Structural Attributes.

    procedure AS_ACTUAL           (t: in out TREE; v: in TREE);
    function  AS_ACTUAL           (t: in TREE) return TREE ; -- assoc
    ...

    -- followed by functions and procedures for about 100 attr utes .....

private

    -- To be filled in...

end Diana;
```

Figure 4-1: Sketch of the DIANA Package

type TREE — An object of this private type is a node of the DIANA structure.

type SEQ_TYPE — An object of this private type is a sequence of nodes of the same class.

type NODE_NAME — This is an enumeration type providing an enumeration literal for each kind of DIANA node.

function MAKE — This function creates and returns a DIANA node of the kind which is its argument. Note that it is overloaded so as also to be able to create an empty list.

procedure DESTROY — This procedure indicates that a node is no longer required.

function KIND — Given a node, this function returns its node-kind.

type ARITIES — This enumeration type provides a literal for each number of structural children a node might have.

function SON_k — For k = 1, 2, 3, each such function returns the k^{th} offspring of a node.

procedure SON_k — For k = 1, 2, 3, each such procedure stores a new k^{th} offspring of the node.

list processing — A collection of functions and procedures implement the usual list-processing primitives.

attributes — For each possible attribute, there is a function to return the value of that attribute at a node, and a procedure to store a new value for the attribute.

A complete listing of the entire DIANA package specification concludes this chapter.

```
with USERPK; use USERPK;
-- Package USERPK provides the following items (see page 77):

-- source_position:   Defines source position in original source program.
--                    Used for error messages.
-- symbol_rep:        Representation of identifiers, strings and characters.
-- value:             Implementation defined.
--                        Gives value of an expression.
--                        Can indicate that no value is computed
-- operator:          Enumeration type for all operators.
-- number_rep:        Representation of numeric literals.
-- comments:          Representation of comments from source program.

package Diana is

   type TREE is private;

   type SEQ_TYPE is private;
```

```
type NODE_NAME is
    (   dn_abort,               dn_accept,              dn_access,
        dn_address,             dn_aggregate,           dn_alignment,
        dn_all,                 dn_allocator,           dn_alternative,
        dn_alternative_s,       dn_and_then,            dn_argument_id,
        dn_array,               dn_assign,              dn_assoc,
        dn_attr_id,             dn_attribute,           dn_attribute_call,
        dn_binary,              dn_block,               dn_box,
        dn_case,                dn_choice_s,            dn_code,
        dn_comp_id,             dn_comp_rep,            dn_comp_rep_s,
        dn_comp_unit,           dn_compilation,         dn_cond_clause,
        dn_cond_entry,          dn_const_id,            dn_constant,
        dn_constrained,         dn_context,             dn_conversion,
        dn_decl_s,              dn_def_char,            dn_def_op,
        dn_deferred_constant,   dn_delay,               dn_derived,
        dn_dscrmt_aggregate,    dn_dscrmt_id,           dn_dscrmt_var,
        dn_dscrmt_var_s,        dn_dscrt_range_s,       dn_entry,
        dn_entry_call,          dn_entry_id,            dn_enum_id,
        dn_enum_literal_s,      dn_exception,           dn_exception_id,
        dn_exit,                dn_exp_s,               dn_fixed,
        dn_float,               dn_for,                 dn_formal_dscrt,
        dn_formal_fixed,        dn_formal_float,        dn_formal_integer,
        dn_function,            dn_function_call,       dn_function_id,
        dn_generic,             dn_generic_assoc_s,     dn_generic_id,
        dn_generic_param_s,     dn_goto,                dn_id_s,
        dn_if,                  dn_in,                  dn_in_id,
        dn_in_op,               dn_in_out,              dn_in_out_id,
        dn_index,               dn_indexed,             dn_inner_record,
        dn_instantiation,       dn_integer,             dn_item_s,
        dn_iteration_id,        dn_label_id,            dn_labeled,
        dn_loop,                dn_l_private,           dn_l_private_type_id,
        dn_membership,          dn_name_s,              dn_named,
        dn_named_stm,           dn_named_stm_id,        dn_no_default,
        dn_not_in,              dn_null_access,         dn_null_comp,
        dn_null_stm,            dn_number,              dn_number_id,
        dn_numeric_literal,     dn_or_else,             dn_others,
        dn_out,                 dn_out_id,              dn_package_body,
        dn_package_decl,        dn_package_id,          dn_package_spec,
        dn_param_assoc_s,       dn_param_s,             dn_parenthesized,
        dn_pragma,              dn_pragma_id,           dn_pragma_s,
        dn_private,             dn_private_type_id,     dn_proc_id,
        dn_procedure,           dn_procedure_call,      dn_qualified,
        dn_raise,               dn_range,               dn_record,
        dn_record_rep,          dn_rename,              dn_return,
        dn_reverse,             dn_select,              dn_select_clause,
        dn_select_clause_s,     dn_selected,            dn_simple_rep,
        dn_slice,               dn_stm_s,               dn_string_literal,
        dn_stub,                dn_subprogram_body,     dn_subprogram_decl,
        dn_subtype,             dn_subtype_id,          dn_subunit,
        dn_task_body,           dn_task_body_id,        dn_task_decl,
        dn_task_spec,           dn_terminate,           dn_timed_entry,
        dn_type,                dn_type_id,             dn_universal_fixed,
        dn_universal_integer,   dn_universal_real,      dn_use,
        dn_used_bltn_id,        dn_used_bltn_op,        dn_used_char,
        dn_used_name_id,        dn_used_object_id,      dn_used_op,
        dn_var,                 dn_var_id,              dn_variant,
        dn_variant_part,        dn_variant_s,           dn_void,
        dn_while,               dn_with
    );
```

```
-- Tree constructors.

function  MAKE     (c: in NODE_NAME)       return TREE;
procedure DESTROY  (t: in TREE);
function  KIND     (t: in TREE)            return NODE_NAME;

-- Tree traversers from the Ada Formal Definition.

type ARITIES is (nullary, unary, binary, ternary, arbitrary);

function  ARITY    (t: in TREE)            return ARITIES;
function  SON1     (t: in TREE)            return TREE;
procedure SON1     (t: in out TREE; v: in TREE);
function  SON2     (t: in TREE)            return TREE;
procedure SON2     (t: in out TREE; v: in TREE);
function  SON3     (t: in TREE)            return TREE;
procedure SON3     (t: in out TREE; v: in TREE);

-- Handling of list constructs.

function  HEAD    (l: in SEQ_TYPE)        return TREE;       -- LISP CAR
function  TAIL    (l: in SEQ_TYPE)        return SEQ_TYPE;   -- LISP CDR
function  MAKE                            return SEQ_TYPE;
                                                     -- return empty list
function  IS_EMPTY (l: in SEQ_TYPE)       return BOOLEAN;
function  INSERT (l: in out SEQ_TYPE;
                  i: in TREE)             return SEQ_TYPE;
                                               -- inserts i at start of l
function  APPEND (l: in out SEQ_TYPE;
                  i: in TREE)             return SEQ_TYPE;
                                               -- inserts i at end of l
```

```
-- Handling of LIST attribute of list constructs.

procedure  LIST     (t: in out TREE; v: in SEQ_TYPE);
function   LIST     (t: in TREE)         return SEQ_TYPE;
                          -- aggregate          has Seq Of COMP_ASSOC
                          -- alternative_s      has Seq Of ALTERNATIVE
                          -- choice_s           has Seq Of CHOICE
                          -- compilation        has Seq Of COMP_UNIT
                          -- comp_rep_s         has Seq Of COMP_REP
                          -- context            has Seq Of CONTEXT_ELEM
                          -- decl_s             has Seq Of DECL
                          -- dscrmt_aggregate   has Seq Of COMP_ASSOC
                          -- dscrt_range_s      has Seq Of DSCRT_RANGE
                          -- enum_literal_s     has Seq Of ENUM_LITERAL
                          -- exp_s              has Seq Of EXP
                          -- generic_assoc_s    has Seq Of GENERIC_ASSOC
                          -- generic_param_s    has Seq Of GENERIC_PARAM
                          -- id_s               has Seq Of ID
                          -- if                 has Seq Of COND_CLAUSE
                          -- inner_record       has Seq Of COMP
                          -- item_s             has Seq Of ITEM
                          -- name_s             has Seq Of NAME
                          -- param_assoc_s      has Seq Of PARAM_ASSOC
                          -- param_s            has Seq Of PARAM
                          -- pragma_id          has Seq Of ARGUMENT
                          -- pragma_s           has Seq Of PRAGMA
                          -- record             has Seq Of COMP
                          -- select_clause_s    has Seq Of SELECT_CLAUSE
                          -- stm_s              has Seq Of STM
                          -- use                has Seq Of NAME
                          -- variant_s          has Seq Of VARIANT
                          -- var_s              has Seq Of VAR
                          -- with               has Seq Of NAME

-- Structural Attributes.

procedure AS_ACTUAL           (t: in out TREE; v: in TREE);
function  AS_ACTUAL           (t: in TREE) return TREE ; -- assoc
procedure AS_ALIGNMENT        (t: in out TREE; v: in TREE);
function  AS_ALIGNMENT        (t: in TREE) return TREE ; -- record_rep
procedure AS_ALTERNATIVE_S    (t: in out TREE; v: in TREE);
function  AS_ALTERNATIVE_S    (t: in TREE) return TREE ; -- case
                                                       -- block
procedure AS_BINARY_OP        (t: in out TREE; v: in TREE);
function  AS_BINARY_OP        (t: in TREE) return TREE ; -- binary
procedure AS_BLOCK_STUB       (t: in out TREE; v: in TREE);
function  AS_BLOCK_STUB       (t: in TREE) return TREE ; -- package_body |
                                                       -- task_body |
                                                       -- subprogram_body
procedure AS_CHOICE_S         (t: in out TREE; v: in TREE);
function  AS_CHOICE_S         (t: in TREE) return TREE ; -- alternative |
                                                       -- named
                                                       -- variant
procedure AS_COMP_REP_S       (t: in out TREE; v: in TREE);
function  AS_COMP_REP_S       (t: in TREE) return TREE ; -- record_rep
procedure AS_CONSTRAINED      (t: in out TREE; v: in TREE);
function  AS_CONSTRAINED      (t: in TREE) return TREE ; -- access | derived
                                                       -- array | subtype
procedure AS_CONSTRAINT       (t: in out TREE; v: in TREE);
function  AS_CONSTRAINT       (t: in TREE) return TREE ; -- constrained
procedure AS_CONTEXT          (t: in out TREE; v: in TREE);
function  AS_CONTEXT          (t: in TREE) return TREE ; -- comp_unit
```

```
procedure AS_DECL_S              (t: in out TREE; v: in TREE);
function  AS_DECL_S              (t: in TREE) return TREE ; -- task_spec
procedure AS_DECL_S1             (t: in out TREE; v: in TREE);
function  AS_DECL_S1             (t: in TREE) return TREE ; -- package_spec
procedure AS_DECL_S2             (t: in out TREE; v: in TREE);
function  AS_DECL_S2             (t: in TREE) return TREE ; -- package_spec
procedure AS_DESIGNATOR          (t: in out TREE; v: in TREE);
function  AS_DESIGNATOR          (t: in TREE) return TREE ; -- subprogram_body
                                                            -- subprogram_decl
                                                            -- assoc | generic
procedure AS_DESIGNATOR_CHAR     (t: in out TREE; v: in TREE);
function  AS_DESIGNATOR_CHAR     (t: in TREE) return TREE ; -- selected
procedure AS_DSCRMT_VAR_S        (t: in out TREE; v: in TREE);
function  AS_DSCRMT_VAR_S        (t: in TREE) return TREE ; -- type
procedure AS_DSCRT_RANGE         (t: in out TREE; v: in TREE);
function  AS_DSCRT_RANGE         (t: in TREE) return TREE ; -- for | reverse
                                                            -- slice
procedure AS_DSCRT_RANGE_S       (t: in out TREE; v: in TREE);
function  AS_DSCRT_RANGE_S       (t: in TREE) return TREE ; -- array
procedure AS_DSCRT_RANGE_VOID    (t: in out TREE; v: in TREE);
function  AS_DSCRT_RANGE_VOID    (t: in TREE) return TREE ; -- entry
procedure AS_EXCEPTION_DEF       (t: in out TREE; v: in TREE);
function  AS_EXCEPTION_DEF       (t: in TREE) return TREE ; -- exception
procedure AS_EXP                 (t: in out TREE; v: in TREE);
function  AS_EXP                 (t: in TREE) return TREE ; -- delay | case
                                                            -- fixed | float
                                                            -- membership | while
                                                            -- address | assign
                                                            -- code | conversion
                                                            -- named | number
                                                            -- qualified
                                                            -- simple_rep
                                                            -- unary | comp_rep
                                                            -- parenthesized
procedure AS_EXP1                (t: in out TREE; v: in TREE);
function  AS_EXP1                (t: in TREE) return TREE ; -- binary | range
procedure AS_EXP2                (t: in out TREE; v: in TREE);
function  AS_EXP2                (t: in TREE) return TREE ; -- range
                                                            -- binary
procedure AS_EXP_CONSTRAINED     (t: in out TREE; v: in
TREE);
function  AS_EXP_CONSTRAINED     (t: in TREE) return TREE ; -- allocator
procedure AS_EXP_S               (t: in out TREE; v: in TREE);
function  AS_EXP_S               (t: in TREE) return TREE ; -- indexed
                                                            -- attribute_call
procedure AS_EXP_VOID            (t: in out TREE; v: in TREE);
function  AS_EXP_VOID            (t: in TREE) return TREE ; -- return
                                                            -- cond_clause
                                                            -- in | in_out | exit
                                                            -- out | record_rep
```

```
procedure AS_GENERIC_ASSOC_S    (t: in out TREE; v: in TREE);
function  AS_GENERIC_ASSOC_S    (t: in TREE) return TREE ; -- instantiation
procedure AS_GENERIC_HEADER     (t: in out TREE; v: in TREE);
function  AS_GENERIC_HEADER     (t: in TREE) return TREE ; -- generic
procedure AS_GENERIC_PARAM_S    (t: in out TREE; v: in TREE);
function  AS_GENERIC_PARAM_S    (t: in TREE) return TREE ; -- generic
procedure AS_HEADER             (t: in out TREE; v: in TREE);
function  AS_HEADER             (t: in TREE) return TREE ; -- subprogram_body
                                                          -- subprogram_decl
procedure AS_ID                 (t: in out TREE; v: in TREE);
function  AS_ID                 (t: in TREE) return TREE ; -- for | attribute
                                                          -- labeled | reverse
                                                          -- named_stm
                                                          -- package_body
                                                          -- package_decl
                                                          -- subtype
                                                          -- task_body
                                                          -- variant_part
                                                          -- type | task_decl
                                                          -- pragma
procedure AS_ID_S               (t: in out TREE; v: in TREE);
function  AS_ID_S               (t: in TREE) return TREE ; -- exception
                                                          -- number | constant
                                                          -- in | in_out
                                                          -- out | var
procedure AS_ITEM_S             (t: in out TREE; v: in TREE);
function  AS_ITEM_S             (t: in TREE) return TREE ; -- block
procedure AS_ITERATION          (t: in out TREE; v: in TREE);
function  AS_ITERATION          (t: in TREE) return TREE ; -- loop
procedure AS_MEMBERSHIP_OP      (t: in out TREE; v: in TREE);
function  AS_MEMBERSHIP_OP      (t: in TREE) return TREE ; -- membership
procedure AS_NAME               (t: in out TREE; v: in TREE);
function  AS_NAME               (t: in TREE) return TREE ; -- accept | address
                                                          -- procedure_call
                                                          -- all | comp_rep
                                                          -- constrained
                                                          -- indexed
                                                          -- instantiation
                                                          -- goto | index
                                                          -- qualified
                                                          -- selected
                                                          -- rename | slice
                                                          -- variant_part
                                                          -- attribute_call
                                                          -- entry_call
                                                          -- record_rep
                                                          -- allocator
                                                          -- assign
                                                          -- attribute | code
                                                          -- conversion
                                                          -- function_call
                                                          -- simple_rep
                                                          -- subunit
```

```
procedure AS_NAME_S          (t: in out TREE; v: in TREE);
function  AS_NAME_S          (t: in TREE) return TREE ; -- abort
                                                         -- with | use
procedure AS_NAME_VOID       (t: in out TREE; v: in TREE);
function  AS_NAME_VOID       (t: in TREE) return TREE ; -- raise | exit
procedure AS_OBJECT_DEF      (t: in out TREE; v: in TREE);
function  AS_OBJECT_DEF      (t: in TREE) return TREE ; -- constant | var
procedure AS_PACKAGE_DEF     (t: in out TREE; v: in TREE);
function  AS_PACKAGE_DEF     (t: in TREE) return TREE ; -- package_decl
procedure AS_PARAM_ASSOC_S   (t: in out TREE; v: in TREE);
function  AS_PARAM_ASSOC_S   (t: in TREE) return TREE ; -- procedure_call
                                                         -- entry_call
                                                      -- pragma
                                                      -- function_call
procedure AS_PARAM_S         (t: in out TREE; v: in TREE);
function  AS_PARAM_S         (t: in TREE) return TREE ; -- procedure
                                                      -- function
                                                      -- entry | accept
procedure AS_PRAGMA_S        (t: in out TREE;  v: in TREE);
function  AS_PRAGMA_S        (t: in TREE) return TREE ; -- comp_unit
procedure AS_RANGE           (t: in out TREE; v: in TREE);
function  AS_RANGE           (t: in TREE) return TREE ; -- integer
                                                      -- comp_rep
procedure AS_RANGE_VOID      (t: in out TREE; v: in TREE);
function  AS_RANGE_VOID      (t: in TREE) return TREE ; -- fixed | float
procedure AS_RECORD          (t: in out TREE; v: in TREE);
function  AS_RECORD          (t: in TREE) return TREE ; -- variant
procedure AS_SELECT_CLAUSE_S (t: in out TREE; v: in TREE);
function  AS_SELECT_CLAUSE_S (t: in TREE) return TREE ; -- select
procedure AS_STM             (t: in out TREE; v: in TREE);
function  AS_STM             (t: in TREE) return TREE ; -- labeled
                                                      -- named_stm
procedure AS_STM_S           (t: in out TREE; v: in TREE);
function  AS_STM_S           (t: in TREE) return TREE ; -- alternative
                                                      -- cond_clause
                                                      -- loop | select
                                                      -- accept | block
procedure AS_STM_S1          (t: in out TREE; v: in TREE);
function  AS_STM_S1          (t: in TREE) return TREE ; -- cond_entry
                                                      -- timed_entry
procedure AS_STM_S2          (t: in out TREE; v: in TREE);
function  AS_STM_S2          (t: in TREE) return TREE ; -- cond_entry
                                                      -- timed_entry
procedure AS_SUBPROGRAM_DEF  (t: in out TREE; v: in TREE);
function  AS_SUBPROGRAM_DEF  (t: in TREE) return TREE ; -- subprogram_decl
procedure AS_SUBUNIT_BODY    (t: in out TREE; v: in TREE);
function  AS_SUBUNIT_BODY    (t: in TREE) return TREE ; -- subunit
procedure AS_TASK_DEF        (t: in out TREE; v: in TREE);
function  AS_TASK_DEF        (t: in TREE) return TREE ; -- task_decl
procedure AS_TYPE_RANGE      (t: in out TREE; v: in TREE);
function  AS_TYPE_RANGE      (t: in TREE) return TREE ; -- membership
procedure AS_TYPE_SPEC       (t: in out TREE; v: in TREE);
function  AS_TYPE_SPEC       (t: in TREE) return TREE ; -- constant | in
                                                      -- in_out | out
                                                      -- var
                                                      -- type
procedure AS_UNIT_BODY       (t: in out TREE; v: in TREE);
function  AS_UNIT_BODY       (t: in TREE) return TREE ; -- comp_unit
procedure AS_VARIANT_S       (t: in out TREE; v: in TREE);
function  AS_VARIANT_S       (t: in TREE) return TREE ; -- variant_part
```

```
-- Lexical Attributes.

procedure LX_COMMENTS        (t: in out TREE;  v: comments);
function  LX_COMMENTS        (t: in TREE)  return comments ;
procedure LX_DEFAULT         (t: in out TREE;  v: Boolean);
function  LX_DEFAULT         (t: in TREE)  return Boolean;
procedure LX_NUMREP          (t: in out TREE;  v: number_rep);
function  LX_NUMREP          (t: in TREE)  return number_rep ;
procedure LX_PREFIX          (t: in out TREE;  v: Boolean);
function  LX_PREFIX          (t: in TREE)  return Boolean;
procedure LX_SRCPOS          (t: in out TREE;  v: source_position);
function  LX_SRCPOS          (t: in TREE)  return source_position ;
procedure LX_SYMREP          (t: in out TREE;  v: symbol_rep);
function  LX_SYMREP          (t: in TREE)  return symbol_rep ;


-- Semantic Attributes.

procedure SM_ACTUAL_DELTA   (t: in out TREE;  v: Float);
function  SM_ACTUAL_DELTA   (t: in TREE)  return Float;
procedure SM_ADDRESS        (t: in out TREE;  v: in TREE);
                                                -- v: EXP_VOID
function  SM_ADDRESS        (t: in TREE)  return TREE ;
                                                -- returns EXP_VOID
procedure SM_BASE_TYPE      (t: in out TREE;  v: in TREE);
                                                -- v: TYPE_SPEC
function  SM_BASE_TYPE      (t: in TREE)  return TREE ;
                                                -- returns TYPE_SPEC
procedure SM_BITS           (t: in out TREE;  v: Integer);
function  SM_BITS           (t: in TREE)  return Integer;
procedure SM_BODY           (t: in out TREE;  v: in TREE);
                                       --      v: SUBP_BODY_DESC,
                                       --         PACK_BODY_DESC,
                                       --         BLOCK_STUB_VOID
function  SM_BODY           (t: in TREE)  return TREE ;
                                       -- returns SUBP_BODY_DESC,
                                       --         PACK_BODY_DESC,
                                       --         BLOCK_STUB_VOID
procedure SM_COMP_SPEC      (t: in out TREE;  v: in
TREE);
function  SM_COMP_SPEC      (t: in TREE)  return TREE ;
procedure SM_CONSTRAINT     (t: in out TREE;  v: in TREE);
                                                -- v: CONSTRAINT
function  SM_CONSTRAINT     (t: in TREE)  return TREE ;
                                                -- returns CONSTRAINT
procedure SM_CONTROLLED     (t: in out TREE;  v: Boolean);
function  SM_CONTROLLED     (t: in TREE)  return Boolean ;
procedure SM_DECL_S         (t: in out TREE;  v: in TREE);  -- v: DECL_S
function  SM_DECL_S         (t: in TREE)  return TREE ;  -- returns DECL_S
procedure SM_DEFN         (t: in out TREE;  v: in TREE);
                                                --          v: DEF_OCCURRENCE
function  SM_DEFN         (t: in TREE)  return TREE ;
                                                -- returns DEF_OCCURRENCE
procedure SM_DISCRIMINANTS  (t: in out TREE;  v: in TREE);    --       v: VAR_S
function  SM_DISCRIMINANTS  (t: in TREE)  return TREE ;    -- returns VAR_S
procedure SM_EXCEPTION_DEF  (t: in out TREE;  v: in TREE);
                                                --         v: EXCEPTION_DEF
function  SM_EXCEPTION_DEF  (t: in TREE)  return TREE ;
                                                -- returns EXCEPTION_DEF
procedure SM_EXP_TYPE       (t: in out TREE;  v: in TREE);    -- v: TYPE_SPEC
function  SM_EXP_TYPE       (t: in TREE)  return TREE ;   -- returns TYPE_SPEC
procedure SM_FIRST          (t: in out TREE;  v: in TREE);    -- v: DEF_OCCURR
function  SM_FIRST          (t: in TREE)  return TREE;   -- returns DEF_OCCURR
```

```
procedure SM_GENERIC_PARAM_S(t: in out TREE;  v: in TREE);
                                                        --       v: GENERIC_PARAM_S
function  SM_GENERIC_PARAM_S(t: in TREE)  return TREE;
                                                        -- returns GENERIC_PARAM_S
procedure SM_INIT_EXP        (t: in out TREE;  v: in TREE);     --     v: EXP_VOID
function  SM_INIT_EXP        (t: in TREE)  return TREE ;    -- returns EXP_VOID
procedure SM_LOCATION        (t: in out TREE;  v: in TREE);     --     v: LOCATION
function  SM_LOCATION        (t: in TREE)  return TREE ;    -- returns LOCATION
procedure SM_NORMALIZED_PARAM_S (t:TREE;   v: in TREE));   --        v: EXP_S
function  SM_NORMALIZED_PARAM_S (t: in TREE) return TREE ; -- returns EXP_S
procedure SM_OBJ_DEF         (t: in out TREE;  v: in TREE));   --  v: OBJECT_DEF
function  SM_OBJ_DEF         (t: in TREE)  return TREE ;  -- returns OBJECT_DEF
procedure SM_OBJ_TYPE        (t: in out TREE;  v: in TREE));    --  v: TYPE_SPEC
function  SM_OBJ_TYPE        (t: in TREE)  return TREE ;   -- returns TYPE_SPEC
procedure SM_OPERATOR        (t: in out TREE;  v: operator);
function  SM_OPERATOR        (t: in TREE)  return operator ;
procedure SM_PACKING             (t: in out TREE;  v: Boolean);
function  SM_PACKING             (t: in TREE)  return Boolean ;
procedure SM_POS          (t: in out TREE;  v: Integer);
function  SM_POS          (t: in TREE)  return Integer ;
procedure SM_REP          (t: in out TREE;  v: Integer);
function  SM_REP          (t: in TREE)  return Integer ;
procedure SM_SIZE         (t: in out TREE;  v: in TREE));      --       v: EXP_VOID
function  SM_SIZE         (t: in TREE)  return TREE ;      -- returns EXP_VOID
procedure SM_SPEC            (t: in out TREE;  v: in TREE);
                                                   -- v: HEADER
                                                   --    GENERIC_HEADER,
                                                   --    PACK_SPEC
function  SM_SPEC            (t:TREE)   return TREE ;  -- returns HEADER
                                                   -- GENERIC_HEADER,
                                                   -- PACK_SPEC
procedure SM_STM          (t: in out TREE;  v: in TREE);     --      v: STM, LOOP
function  SM_STM          (t: in TREE)  return TREE ;     -- returns STM, LOOP
procedure SM_STORAGE_SIZE    (t: in out TREE;  v: in TREE));    --     v: EXP_VOID
function  SM_STORAGE_SIZE    (t: in TREE)  return TREE ;    -- returns EXP_VOID
procedure SM_STUB            (t: in out TREE;  v: in TREE);
function  SM_STUB            (t: in TREE)  return TREE ;
procedure SM_TYPE_SPEC       (t: in out TREE;  v: in TREE);     --   v: TYPE_SPEC
function  SM_TYPE_SPEC       (t: in TREE)  return TREE ;  -- returns TYPE_SPEC
procedure SM_TYPE_STRUCT     (t: in out TREE;  v: in TREE);     -- v: TYPE_SPEC
function  SM_TYPE_STRUCT     (t: in TREE)  return TREE ;  -- returns TYPE_SPEC
procedure SM_VALUE               (t: in out TREE;  v: value);
function  SM_VALUE               (t: in TREE)  return value ;

-- Code Attribute.

procedure CD_IMPL_SIZE       (t: in out TREE;  v: Integer);
function  CD_IMPL_SIZE       (t: in TREE)  return Integer ;

private

    -- To be filled in...

end Diana;
```

CHAPTER 5
EXTERNAL REPRESENTATION OF DIANA

This chapter describes how a DIANA tree may be represented in ASCII for communication between different computing systems. The presentation is informal; for a detailed discussion of the issues involved, see Chapter 4 of the IDL Reference Manual [9]. Although any conforming implementation of DIANA is required to be able to map to and/or from this *external* representation of DIANA, other *internal* representations are permitted. Indeed, we expect these latter (non-conforming) representations to be the preferred means of communication between tools on a single computing system. The standard external form is defined to assist debugging and to allow communication between computing systems, not as the typical communication between tools.

The design of this external representation was guided by three principles:

- There must be a relatively straightforward way of deducing the external representation from the DIANA specification of Chapter 2.
- The external representation must not unduly constrain the implementation options outlined in Chapter 6.
- It must be possible to map between the external representation and a variety of internal representations in a reasonably efficient manner.

We expect that each installation that wishes to communicate with others via an ASCII representation of DIANA will create a reader/writer utility to map between the external representation and whatever internal representations are in use at the installation.

The external representation is described in Figure 5-1 on page 146. It is the usual sort of recursive construction. Note that square brackets [...] surround the attributes of a *node* and angle brackets <...> surround items of a sequence.

We illustrate the external representation using the IDL example from Section 1.4.1, repeated here as Figure 5-2 on page 147. From this example, nodes plus, leaf, and tree might be represented externally as follows:

```
plus            — a node with no attributes

leaf [ name "A"; src representation_of_source_position ] — leaf for A

tree [ left leaf [name "A"]; right leaf [name "B"]; op plus] — A + B
```

DEFINITION OF EXTERNAL DIANA

node represented by the name of its type, followed by '[', followed by the representation of its attributes (separated by semicolons), followed by ']'. If there are no attributes, the '[ ]' may be omitted.

attribute represented by the name of the attribute, followed by the representation of the value of the attribute.

comment start with double hyphen ('--'); terminate with the end of the line.

REPRESENTATION OF BASIC TYPES

`Boolean` represented by the tokens `TRUE` and `FALSE`.

`Integer` represented by a sequence of digits with an optional sign. The value is interpreted as being a decimal integer.

`Rational` represented as a decimal or based number (in the ADA sense and using the ADA syntax), or as the quotient of two unsigned integers, decimal numbers, or based numbers.

`String` represented as the sequence of ASCII characters representing the value of the string, surrounded by double quotes. Any quotes within the string must be doubled. The nonprinting ASCII characters are represented as in ADA.

Sequence represented by a sequence of representations for individual values of the sequence, separated by spaces, and surrounded by angle brackets ('<. . .>').

Private types are provided by the structure definition. For our purposes, the external representations of the private types used in DIANA are provided in a refinement of the DIANA abstract structure.

Spaces are not significant except to separate tokens.

Case distinctions between identifiers (such as node names) are significant, as in IDL.

Figure 5-1: External DIANA Form

Note that no representation is shown for the value of the attribute *src*, which is the private type Source_Position: this point is addressed further below. Note also that, because these examples show external DIANA which is expected to be ASCII text, the usual typographic conventions for node names and attributes are not followed in them.

```
Structure ExpressionTree Root EXP Is

    -- First we define a private type.

    Type Source_Position;

    -- Next we define the notion of an expression, EXP.

    EXP  ::=  leaf | tree ;

    -- Next we define the nodes and their attributes.

    tree  =>  op: OPERATOR,  left: EXP,  right: EXP ;
    tree  =>  src: Source_Position ;
    leaf  =>  name: String ;
    leaf  =>  src: Source_Position ;

    -- Finally we define the  notion of an OPERATOR as the
    -- union of a collection of nodes; the null => productions
    -- are needed to define the node types since
    -- node type names are never implicitly defined.

    OPERATOR  ::=  plus | minus | times | divide ;

    plus => ;   minus => ;   times => ; divide => ;

End
```

Figure 5-2: Example *ExpressionTree* of IDL Notation

The external representation also provides a means for sharing attribute values between nodes. This fact does not necessarily imply that the corresponding internal representation is shared; for some attributes, the sharing in the external representation can be viewed simply as a technique for compressing space.

However, any attribute value which is *inherently shared internally*[1] must be represented externally in shared form. All of the tree-valued attributes of DIANA fall in this category.

In order for an attribute value to be shared in the external representation, one occurrence of the value must be labeled and all other occurrences must refer to that label. Any attribute value may be labeled, including node-valued attributes. The labeled *occurrence* of the value is represented in a normal way, except that it is preceded by a label identifier and a colon (`:'). Each label *reference* consists of the label identifier followed by a caret (`^'), rather than the usual representation of the attribute value. A label identifier is a sequence of letters, digits, and isolated underscores starting, with a letter; case distinctions among the letters are significant. For example, the tree for A+A could be represented in any one of the following four ways (among others):

```
tree [ left leaf [ name "A"]; op plus; right leaf [ name "A" ] ]

tree [ left leaf [ name y: "A" ]; op plus; right leaf [ name y^ ] ]

tree [ left  x:leaf [ name "A"]; op plus; right x^ ]

tree [ left x^; op plus; right x:leaf [ name "A" ] ]
```

Additionally, a node-valued attribute can be written *free standing without* being nested within some other node. For example, a fifth representation for the preceding example is

```
tree [ left x^; op plus; right x^ ]
x: leaf [ name "A" ]
```

Note that in these examples we have consistently avoided giving a representation for the source position attributes. Recall that source position is a private type whose representation must be supplied as part of the structure definition or a refinement of the structure. One way to represent the source position is to use the representation defined in the example refinement in Section 1.4.3 on page 28, repeated here for convenience in Figure 5-3 on page 149. Using this external form, a source position might be represented using the node structure:

```
leaf [ name "A";
       src source_position
         [ file "<user>test.ada" ;  line 3 ;  char 15 ]
     ]
```

---

[1] The phrase 'inherently shared internally' is intentionally loose. We believe that the phrase captures the essence of the situations where it is convenient to use sharing in the external representation. For a complete discussion of this issue, see the IDL Reference Manual.

```
Structure AnotherTree Renames ExpressionTree Is

    -- first the internal representation of Source_Position

    For Source_Position Use Source_Package;

    -- next the external representation of Source_Position
    -- is given by a new node type, source_external_rep

    For Source_Position Use External source_external_rep;

    -- finally, we define the node type source_external_rep

    source_external_rep => file   : String,
                           line   : Integer,
                           char   : Integer;

End
```

Figure 5-3: Example *AnotherTree* of IDL

Alternatively, a specification could define the source position to be represented externally as a string:

```
leaf [ name "A"; src "<user>test.ada/15/3"]
```

Each of these particular external representations in some sense contains the same information in that either one could be mapped to the same internal representation by the reader utility. Each installation must establish conventions for communicating between the reader/writer utility and its user-supplied packages to allow such user-supplied types to be mapped to and from the external form. Of course, other representations for the source position attribute are possible, many containing quite different information. A more complete treatment of the external representation of private types may be found in the IDL Reference Manual.

The refinement of the DIANA structure defines the external representation for four private types, *symbol_rep*, *number_rep*, *operator*, and *value*. Types *symbol_rep*, and *number_rep* are represented as strings externally, and *operator* is represented by an enumeration type.

The type *symbol_rep* is a string that contains the source representation of identifiers. The type *symbol_rep* also represents character literals, which are distinguished from other identifiers by surrounding the character with single quote

marks, as in ADA. An implementation must decide how to treat upper and lower case characters: it can normalize the representations of identifiers to use the basic character set, all lower case letters changed to upper case, or it can preserve the case used in the source, so that source can be reconstructed accurately.

The type *number_rep* is a string that has the source representation of numeric literals. An implementation may choose to normalize *numeric_literals* by removing underscores.

The type *operator* is represented by an enumeration type. In the refined DIANA specification a minimum enumeration set is given; it may be expanded by an implementation to include any other built-in subprograms.

The type *value* is represented as an integer or rational type if a value has been computed, or with a distinguishing node for the cases where the value has not yet been computed. A representation for ADA strings and arrays is also provided: a sequence of values.

A complete external representation starts with an indication of the root node of the corresponding structure, followed by a sequence of zero or more representations of nodes. The root indication can be either a label referencing a node elsewhere in the external representation or the root node itself. Since the representation of subnodes can be contained within the representation of the parent node, it is possible for the entire external representation to be given by the *root* (a compilation node in DIANA). It is permissible, on the other hand, to represent the DIANA tree in a flat form, where node-valued attributes are always represented by labels referencing non-nested representations of the nodes.

Following are two examples, both in flat form. In each case a short ADA fragment is followed by the external form of the DIANA. Note that these examples, like the figures in Chapter 3, are incomplete in that some attributes are omitted for expository convenience.

```
-- From package STANDARD (sort of)
type BOOLEAN is (FALSE, TRUE);
type INTEGER is range MIN_INT .. MAX_INT;

----------

PD0:  type              [ as_id PD1^ ;
                          as_var_s PD2^ ;
                          as_type_spec PD3^ ]
PD1:  type_id           [ lx_symrep "BOOLEAN" ;
                          sm_type_spec PD3^ ]
PD2:  var_s             [ as_list < > ]
PD3:  enum_literal_s    [ as_list < PD4^ PD5^ >
                          sm_size void ]
PD4:  enum_id           [ lx_symrep "FALSE" ;
                          sm_obj_type PD3^ ;
                          sm_rep 0 ;
                          sm_pos 0 ]
PD5:  enum_id           [ lx_symrep "TRUE" ;
                          sm_obj_type PD3^ ;
                          sm_rep 1 ;
                          sm_pos 1 ]
PD6:  type              [ as_id PD8^ ;
                          as_var_s PD7^ ;
                          as_type_spec PD9^ ]
PD7:  var_s             [ as_list < > ]
PD8:  type_id           [ lx_symrep "INTEGER" ;
                          sm_type_spec PD9^ ]
PD9:  integer           [ as_range PD10^ ;
                          sm_type_struct PD9^ ;
                          sm_size void ]
PD10: range             [ as_exp1 PD11^ ;
                          as_exp2 PD12^ ;
                          sm_base_type PD9^ ]
PD11:  used_object_id   [ lx_symrep "MIN_INT" ;
                          sm_defn xxx^ ;              -- def for MIN_INT
                          sm_value xxx^ ;             -- def for MIN_INT
                          sm_exp_type PD9^ ]
PD12:  used_object_id   [ lx_symrep "MAX_INT" ;
                          sm_defn xxx^ ;              -- def for MAX_INT
                          sm_value xxx^ ;             -- def for MAX_INT
                          sm_exp_type PD9^ ]
```

```
package REPORT is
    function EQUAL ( X, Y : INTEGER ) return BOOLEAN;
end REPORT;

----------

A01: comp_unit          [ as_pragma_s A02^ ;
                          as_context  A03^ ;
                          as_unit_body A04^ ]
A02: pragma_s           [ as_list < > ]
A03: context            [ as_list < > ]
A04: package_decl       [ as_id A05^ ;
                          as_package_def A06^ ]
A05: package_id         [ lx_symrep "REPORT" ;
                          sm_spec A06^ ;
                          sm_body void ;
                          sm_address void ]
A06: package_spec       [ as_decl_s1 A08^ ;
                          as_decl_s2 A07^ ]
A07: as_decl_s          [ as_list < > ]
A08: as_decl_s          [ as_list <  A09^ > ]
A09: subprogram_decl    [ as_designator A10^ ;
                          as_header A11^ ;
                          as_subprogram_def void ]
A10: function_id        [ lx_symrep "EQUAL" ;
                          sm_spec A11^ ;
                          sm_body void ;
                          sm_location void ]
A11: function           [ as_param_s A12^ ;
                          as_name A18^ ]
A12: param_s            [ as_list < A13^ > ]
A13: in                 [ as_id_s A14^ ;
                          as_name A17^ ;
                          as_exp_void void ]
A14: id_s               [ as_list < A15^ A16^ > ]
A15: in_id              [ lx_symrep "X" ;
                          sm_init_exp void ;
                          sm_obj_type PD9^ ]
A16: in_id              [ lx_symrep "Y" ;
                          sm_init_exp void ;
                          sm_obj_type PD9^ ]
A17: used_name_id       [ lx_symrep "INTEGER" ;
                          sm_defn PD8^ ;
A18: used_name_id       [ lx_symrep "BOOLEAN" ;
                          sm_defn PD1^ ]
```

CHAPTER 6
IMPLEMENTATION OPTIONS

One obvious implementation of a compiler using the DIANA intermediate form is to produce the complete DIANA abstract tree as the result of semantic analysis, representing each abstract tree node by a variant record on a heap and using pointers to implement those attributes that reference other nodes. In some applications such an implementation may be completely appropriate; in others, it may not. The purpose of this chapter is to illustrate some other implementation options that are possible. We cannot, of course, describe all conceivable options; our goal is merely to describe enough of them to make the point that the obvious implementation is not the only possible one.

At the risk of repeating the point once too often, we stress that DIANA is *representation independent*. Possible implementations include any of the schemes mentioned below, many others, and combinations of them. Each possibility makes good sense in certain applications or for certain implementation environments.

A Coroutine Organization: The Front and Back Ends of the compiler might be organized in a *coroutine* manner, in which the Front End produces a portion of the intermediate form after which the Back End produces code for this portion and then discards the unneeded pieces of its input. In this organization there would never be a DIANA representation of the entire compilation unit at any one time. Instead, only a consistent DIANA subtree for the portion being communicated is needed. Although this type of organization may limit the amount of optimization that can be done, it is often useful and is completely consistent with the DIANA model. To use this style of compiler organization, the user needs only to ensure that the values of all of the attributes for the portion of the tree being communicated are defined properly.

Non-Tree Structures: Many simple compilers use a linear representation, such as Polish postfix, for the intermediate form. Such a representation has the advantage of simplifying certain tree traversals, and indeed may be obtained from the DIANA tree by just such a traversal. Such representations may also have an advantage in that they are more efficient where storage is limited or paging overheads are high. Again, such representations are fully within the spirit of DIANA. Where DIANA requires a (conceptual) pointer, it may be replaced by an

index into the linear representation.

DAG Representation: The structural attributes of DIANA define a tree corresponding to the abstract syntax of ADA. So long as the processing algorithms do not require distinct copies of identical subtrees, such subtrees may be *shared* to save memory space. The resulting storage structure is a directed acyclic graph, or DAG. This observation is especially important with respect to leaves of the tree and to certain attribute values. Typically, for example, about half the nodes in a tree are leaves; thus, substantial space can be saved by using a single instance of a **used_name_id** node to represent all of its logical occurrences in the DIANA tree. Similarly, occurrences of the attributes that represent literal values and the string name of identifiers in the program can be pooled.

Attributes Outside the Nodes: There is no need for the attributes of a node to be stored contiguously. As there are many variations on this theme, we illustrate just one here. Suppose that the general storage representation to be used involves storing each node as a record in the heap and using pointers to encode the structural attributes. Because there are a number of different attributes associated with each node type, one may not wish to store these attributes directly in the records representing the nodes. Instead, one might define a number of vectors (of records) where the records in each vector are tailored to the various groupings of attribute types in DIANA nodes. Using this scheme, the nodes themselves need contain only an index into the relevant vector. Such a scheme has the advantage of making nodes of uniform size as well as facilitating the sharing of identical sets of attribute values.

General Set of Attributes: All nodes can be implemented with a general set of attributes, and all other attributes could be kept outside the nodes. A Boolean-valued attribute in the node can then be used to indicate that an attribute outside the node exists. This method is useful for attributes that may be on several nodes but is generally void (such as *lx_comments*).

Nodes Inside Other Nodes: Although an attribute of a node may reference another node, there is no implication that a pointer is required; the referenced node may be directly included in the storage structure of the outer node so long as the processing permits this. This approach is especially important where the referenced node has no attributes. For example, the **binary** node of DIANA has an attribute called *as_binary_op* which references one of a number of possible nodes—all of which have no attributes. In effect, this attribute's value is an

enumeration type and can be implemented as a small integer stored in the binary node's storage area.

Copies of Attribute Values: An implementation may choose to copy the value of an attribute, *e.g.*, if the attribute value is stored in another compilation unit. The implementation must, of course, preserve the semantics of the equality test and assignment operations for attribute values, as discussed in Section 3.9.

Separate Symbol Tables: The collection of nodes types which constitute DEF_OCCURRENCEs are effectively a symbol table. This presentation discusses such nodes as if they were part of the tree, but an implementation may elect to collect these nodes together into a compact structure, physically separate from the tree.

APPENDIX I
THE PREDEFINED ENVIRONMENT

The semantics of ADA provide that an ADA program may reference certain entities that are not defined within the program itself. There are four cases:

universal types These cannot be mentioned by the programmer but are referenced only implicitly. For example, they are referenced in describing the type of a number, or in describing the result of certain ADA attributes.

predefined language environment
This is essentially the package STANDARD.

attributes These include both those predefined by ADA as well as those defined by the implementation.

pragmas These include both those predefined by ADA as well as those defined by the implementation.

In the following four sections of this appendix we describe how the DIANA form for each of these is derived.

I.1. Universal Types

The notion of universal types is used in ADA to associate a type with a number declaration and to define the result type of certain attributes. To represent these notions, DIANA extends the class TYPE_SPEC by three nodes:

```
TYPE_SPEC ::=        universal_integer |
                     universal_real |
                     universal_fixed ;
```

These nodes, which have no attributes, can be referenced only by semantic attributes of a program; they never appear directly in the tree. The type **universal_real** covers both fixed and float types in cases where they cannot be distinguished, as in number declarations.

I.2. The Predefined Language Environment

The predefined environment of ADA is specified by the package STANDARD, given in Appendix C of the ADA LRM. The DIANA tree for it may be obtained by simply compiling this package with a Front End, though the compilation must be

done in a special mode since some attributes must be determined by special rules. In a few cases (such as *cd_impl_size* for numeric types), the attributes must be explicitly assigned; they cannot be derived from any further environment inquiry. Note that this operation need be done only once; the DIANA form can then be preloaded into all programs that process the DIANA form of ADA.

Since the Front End and Back End must be able to agree on the *operator* type (see Section 3.8.5) and the Front End must be able to communicate this information to the Back End, the two must agree on how the representation of package STANDARD is to be augmented to include this information.

I.3. Attributes

Appendix A of the ADA LRM describes a set of predefined language attributes; these may be extended by an implementation, see LRM Section 4.1.4. DIANA requires a unique definition point for each of these attribute identifiers. DIANA does not define additional information for checking that attributes are used correctly; the design of this information is a choice for each implementation. We also need a string representation of the attribute name (to reconstruct the source, for example). The resulting structure looks like:

```
DEF_ID ::=          attr_id;
attr_id =>          lx_symrep          : symbol_rep;
```

The complete definition of an ADA program requires nodes for all the implementation supported attributes; these are easily constructed. Using the external form of DIANA defined in Chapter 5, for example, two of the predefined attribute nodes are:

```
attr_id [ lx_symrep "BITS" ]
attr_id [ lx_symrep "SMALL" ]
```

I.4. Pragmas

Appendix B of the ADA LRM lists the language-defined pragmas for ADA. An implementation is free to expand this set by defining additional pragmas. DIANA provides a definition point for the identifiers needed to represent the complete set of pragmas known to an implementation. The DIANA representation of these is similar to its representation of attributes described above; in the predefined environment, diana provides the information necessary to identify the pragma

names and their names of its arguments. In addition, where the possible values of a pragma's arguments are named (*e.g.*, for pragma LIST the values "ON" and "OFF"), a defining occurrence for the names of the values is also provided.

The defining occurrence for an identifier used in conjunction with a pragma in DIANA has the following structure:

```
DEF_ID ::=          pragma_id | ARGUMENT ;
pragma_id =>        as_list          : Seq Of ARGUMENT ;
ARGUMENT ::=        argument_id ;
pragma_id =>        lx_symrep        : symbol_rep ;
argument_id =>      lx_symrep        : symbol_rep ;
```

A list of argument names is introduced for those situations where multiple argument names are possible, as for example for the various check names for the SUPPRESS pragma. Note that the list is also used to introduce the names of the values the pragma's arguments may take.

As with the attributes, an implementation must supply a set of nodes for the various language-defined and implementation-defined pragmas. Here are two examples in external DIANA form:

```
pragma_id [ lx_symrep "LIST"; as_list <L1^ L2^>]
L1: argument_id [ lx_symrep "ON" ]
L2: argument_id [ lx_symrep "OFF"]

pragma_id [ lx_symrep "PRIORITY" ]
```

All checks concerning the correct use of a pragma are assumed to have been done during semantic analysis, and performing these checks will necessarily require knowledge of the semantics the pragma that DIANA cannot supply. The predefined environment merely provides the defining occurrences for the identifiers used.

For language-defined pragmas, DIANA requires that the pragma subtree represents a correct pragma: that is, for each pragma the proper semantic checking has been done. For pragmas not supported by an implementation DIANA requires that the *structure* of the pragma subtree is present and contains the lexical information but does not require that the semantic attributes are correct. In most cases this requirement means the pragma name and argument names are represented by **used_name_id** *nodes* whose *sm_defn* attribute is void.

There are several situations where the arguments to a pragma are types or objects defined by the user. The **pragma** node has a structural attribute which represents the list of actual arguments to a particular pragma; the list in the **pragma_id** corresponds in a sense to formal parameters. Figure I-1 shows the tree for the fragment

```
type C is array(1..10) of CHARACTER;
pragma PACK(C);
```

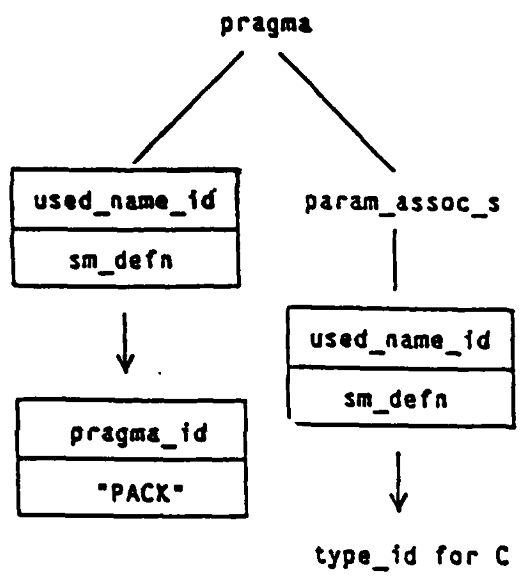


Figure I-1: Example of a Pragma

APPENDIX II
THE ABSTRACT PARSE TREE

ADA's Formal Definition assumes a parse tree that is structurally quite similar to the DIANA tree described in Chapter 2. This appendix shows the IDL representation for this parse tree.

Following are the principal differences between the parse tree and the DIANA tree:

- The parse tree has *no* semantic or code attributes.

- The parse tree has **apply** nodes instead of function call, procedure call, entry call, attribute call, indexed, conversion, and slice nodes.

IDL provides a means for deriving a structure from a previously defined structure by describing the new structure in terms of *changes* or *edits* to the old structure. This form of structure declaration has the following basic form:

```
Structure new_structure Root root
From old_structure Is
    -- "edits" to old structure
End
```

There are two sorts of edits: additions to the original structure and deletions from it. Additions are indicated by simply *including the additions* within the structure declaration as normal IDL definitions. Deletions are indicated by clauses beginning with the keyword **Without**, followed by a list of items to be deleted from the original structure in forming the new one. Five kinds of deletions can be made:

- Deletion of a particular element from the right side of a class definition is indicated by an entry of the form

  ```
  class_name ::= element_name
  ```

  Here an "element" can be either a class or a node. Here is an example:

  ```
  -- old
      EXP ::= foo | leaf | tree
  -- without clause
      Without EXP ::= leaf
  -- new
      EXP ::= foo | tree
  ```

- Deletion of a particular attribute from the right side of a node definition is indicated by a line of the form

```
node_name => attribute_name
```

Here is an example:

```
-- old
   tree => left:EXP, op:OP, right:EXP
-- without clause
   Without tree => left
-- new
   tree => op:OP, right:EXP
```

- Deletion of an entire class definition is indicated by giving just the class name followed by `::=', as in

  ```
  FOO ::=
  ```

- Deletion of an entire node definition is indicated by giving just the node name followed by `=>', as in

  ```
  foo =>
  ```

- Deletion of an attribute name is indicated by writing

  ```
  * => foo
  ```

  The attribute is thereby deleted from all nodes which named it.

Using this notation, we now derive from Diana the structure ParseTree, with root COMPILATION.

```
Structure ParseTree Root COMPILATION From Diana Is

-- ParseTree has APPLY nodes instead of function call, procedure call,
-- entry call, attribute call, indexed, conversion, and slice nodes.

Without
    function_call =>,
    procedure_call =>,
    entry_call =>,
    attribute_call =>,
    indexed =>,
    slice =>,
    conversion => ,

    NAME ::=                attribute_call,
    NAME ::=                function_call,
    NAME ::=                slice,
    NAME ::=                indexed,
    EXP ::=                 conversion,

    STM ::=                 procedure_call,
    STM ::=                 entry_call;

-- additions for APPLY

    STM ::=                 NAME;
    NAME ::=                apply;

    apply =>                as_name             : NAME,
                            lx_srcpos           : source_position,
                            lx_comments         : comments,
                            as_param_assoc_s    : GENERAL_ASSOC_S;

    GENERAL_ASSOC_S ::= general_assoc_s;
    general_assoc_s =>      as_list             : Seq Of GENERAL_ASSOC,
                            lx_srcpos           : source_position,
                            lx_comments         : comments,
    GENERAL_ASSOC ::=       ACTUAL | RANGE | named;

-- ParseTree has only one kind of USED_ID

Without
    used_name_id => ,
    used_object_id => ,
    used_number_id =>,
    used_bltn_id =>,
    USED_ID ::=             used_name_id,
    USED_ID ::=             used_object_id,
    USED_ID ::=             used_bltn_id;

-- additions for USED_ID

    USED_ID ::=             used_id;
    used_id =>              lx_srcpos           : source_position,
                            lx_comments         : comments,
                            lx_symrep           : symbol_rep;

-- ParseTree has only one kind of USED_OP

Without
    used_op => ,
    string_literal => ,
    used_bltn_op => ,
    DESIGNATOR ::=          used_op,
    EXP ::=                 string_literal;

-- additions for USED_OP

    DESIGNATOR ::=          used_string;
    used_string =>          lx_srcpos           : source_position,
                            lx_comments         : comments,
                            lx_symrep           : symbol_rep;
```

```
-- ParseTree has no semantic attributes

Without
* => sm_actual_delta,
* => sm_address,
* => sm_base_type,
* => sm_bits,
* => sm_body,
* => sm_comp_spec,
* => sm_constraint,
* => sm_controlled,
* => sm_decl_s,
* => sm_defn,
* => sm_discriminants,
* => sm_exception_def,
* => sm_exp_type,
* => sm_first,
* => sm_generic_param_s,
* => sm_init_exp,
* => sm_location,
* => sm_normalized_comp_s,
* => sm_normalized_param_s,
* => sm_obj_def,
* => sm_obj_type,
* => sm_operator,
* => sm_packing,
* => sm_pos,
* => sm_record_spec,
* => sm_rep,
* => sm_size,
* => sm_spec,
* => sm_stm,
* => sm_storage_size,
* => sm_stub,
* => sm_type_spec,
* => sm_type_struct,
* => sm_value;

-- ParseTree has no code attributes

Without
* => cd_impl_size;

End
```

APPENDIX III
RECONSTRUCTING THE SOURCE

One of the basic principles of DIANA is that the structure of the original source program is to be retained in the DIANA representation. DIANA has been designed so that the front end of an ADA compiler (or any other tool that produces DIANA from ADA) can include in the DIANA sufficient information so that, to a first approximation at least, the original ADA text can be recreated from the DIANA. This ability enables an APSE tool such as a pretty-printer to work directly from the DIANA form, or a syntax-directed editor to operate directly on it. The DIANA form can stand alone without reference to the original source; some APSE designs might elect to discard the source and keep just the DIANA form, using a pretty-printer when a source listing is required.

DIANA's design deliberately includes certain *normalizations* of source programs. These are omissions from the DIANA of enough information to reconstruct the original program *exactly*, and the effect of omitting these data is that the reconstructed source program is of necessity *normalized* in certain ways. (The normalizations are discussed in Section III.3.) *Although the information* lost by making these normalizations could be retained by providing additional lexical attributes, DIANA's design is predicated on the assumption that the value to the user of this information does not justify imposing on all DIANA users the cost in processing time to record the additional data or in space to store them.

III.1. General Principles

The structure of DIANA's original design followed the Abstract Syntax Tree (AST) of the ADA Formal Definition (AFD), which was designed to include adequate information to permit source reconstruction. Unfortunately, the AFD is based on ADA-80, and DIANA's (present) design is based on the syntax of on ADA-82, which differs from that of ADA-80 in important ways.

In Chapter 2, we showed the connection between the ADA-82 syntax and the DIANA structure by including the former with the description of the corresponding nodes and attributes. There is a close correspondence between ADA's syntax and DIANA's structural attributes, as shown in the examples in the next section. It is this correspondence that permits source reconstruction.

The discussion on formalization of DIANA in Section 1.1.4 on page 14 is also relevant. Any technique to solve the problem addressed in that section will shed light on source reconstruction.

III.2. Examples

A few examples illustrate the reconstruction process. Consider first the ADA assignment statement, with syntax and DIANA structural attributes as follows:

```
Ada Syntax (Section 5.2 of the Ada LRM):
    assignment_statement ::=
        name := expression ;

Diana rules:
    assign => as_name   : NAME,
              as_exp    : EXP;
```

The ADA text corresponding to an assign node is thus the text that led to the NAME (i.e., the value of the *as_name* attribute), followed by `:=', followed by the text that led to the EXP (i.e., the value of the *as_exp* attribute, followed by `;'. We can summarize by writing that the source text for an assign node is

```
<NAME> := <EXP> ;
```

Here the angle brackets (<>) indicate that the the text for the corresponding subtrees must be filled in.

As a second example, consider an ADA block:

```
Ada Syntax (Section 5.6 of the Ada LRM):
    block_statement ::=
      [block_simple_name]
        [declare
            declarative_part]
         begin
            sequence_of_statements
        [exception
            exception handler {exception_handler}]
         end [block_simple_name] ;

Diana rules:
    block => as_item_s          : ITEMS_S,
             as_stm_s           : STM_S,
             as_alternative_s   : ALTERNATIVE_S;
```

Thus for an unlabeled block node used as a statement the following text is generated.

```
declare
   <item_s>
begin
   <stm_s>
exception
   <alternative_s>
end;
```

In a few places the text to be generated depends on the structural child. In the block statement example, it is important that the text **exception** be generated only when the sequence of alternatives is non-empty (*i.e.*, the *as_alternative_s* child is empty), since the syntax of ADA-82 requires at least one exception handler after the word **exception**. (ADA-80 permitted an empty list of handlers.) Similarly, a private part should be generated only for a package that contains a non-empty list of private declarations.

In a similar vein, sometimes the text to be generated depends on the structural parent. Again the **block** node provides a good example. When **block** appears as the descendant of a **subprogram_body** node, the word **declare** should not be generated.

III.3. Normalizations of the Source

A *normalization* of the source is a deliberate omission from the DIANA structure of information that would be required to produce an *exact* recreation of the source text. Most of the normalizations are imposed by the AFD. DIANA includes the following normalizations:

- The optional identifiers following the reserved word **end** are not represented in DIANA. This decision means that during reconstruction the program is normalized either always to include the end labels or always to omit them.

- DIANA does not require that extra spaces between lexical tokens be preserved.

- Variant spelling of an identifier, as for example "FOO" and "Foo" and "foo", need not be recorded in DIANA. This is a lexical issue that does not affect the semantics.

- Alternate writings of numeric constants need not be preserved. For example, in

  ```
  2   002   0_0_2
  2#1111_1111#   16#FF#   016#0FF#   255
  12e1   1.2e2   0.12e+3   01.2e02
  ```

  all the values on each line would be represented identically in the

DIANA and so would be reconstructed identically. This issue is essentially the same as the variant spelling of identifiers; DIANA does not require that variations be preserved.

A few normalizations of the AFD are no longer in DIANA, because of changes in ADA-82's syntax from the ADA-80 syntax used in the AFD.

- In the AFD (and therefore in the original design of DIANA), all infix operators (except the short circuit and membership operators) are converted to function calls. That is, each of

  ```
  X + Y
  "+"(X, Y)
  ```

  gave rise to the same DIANA structure. Thus the original program could not be reconstructed, since it could not be determined whether the original had an infix or prefix form for these operators. ADA-82 requires that this distinction be maintained to meet the conformance rules for initial values of *default formal parameters*, stated in ADA LRM Section 6.3.1.

- In formal parameter declarations for subprograms, the mode in is optional. Originally, the presence of the word in in a formal part was *not* recorded in the DIANA. The conformance rules of Section 6.3.1 requires that this information be maintained.

- The AFD omits **parenthesized** nodes if the parentheses are redundant. The conformance rules just referred to require retention of these nodes.

III.4. Comments

In order properly to reconstruct the source, DIANA must be capable of recording comments. To this end, every DIANA node that has a source position attribute (*i.e.*, all those which correspond to points in the source program) has the additional attribute

lx_comments : **comments**;

which is an implementation-dependent type. The implementation may choose how accurately comment positions are recorded and how to associate comments with particular nodes.

The way a user chooses to comment a program greatly affects the ability of any internal representation to make a meaningful association of comments to nodes. When there is a coding standard that enforces a commenting style, assumptions can be made that make the association easier. Since standards such as these are likely to be only enforced locally, comments are treated as an implementation-dependent type. DIANA makes no requirement about either the

internal or the external representation of comments. and an implementation does not have to support the *lx_comments* attribute to be considered a DIANA *producer* or DIANA *consumer*.

One method for attaching comments to tree nodes is described in [1]. It distinguishes between comments preceding or following the subtree which is represented by the node.

APPENDIX IV
DIANA SUMMARY

This appendix contains a list of all the class and node definitions sorted by the name of the class or node. Class definitions are given first; all class names are upper case. Node definitions follow; node names are lower case. With each definition is listed the section number and page number within Chapter 2 where the corresponding concrete syntax can be found.

```
ACTUAL ::= EXP;                                  6.4        61
ALIGNMENT ::= alignment;                         13.4.A     74
ALTERNATIVE ::= alternative |                    5.4        53
                pragma;
ALTERNATIVE_S ::= alternative_s;                 5.4        53
ARGUMENT ::= argument_id;                        App. I     76
BINARY_OP ::= SHORT_CIRCUIT_OP;                  4.4.A      48
BLOCK_STUB ::= block;                            6.3        60
BLOCK_STUB ::= stub;                             10.2.B     70
BLOCK_STUB_VOID ::= block |                      9.1.A      65
                    stub |
                    void;
CHOICE ::= EXP |                                 3.7.3.B    43
           DSCRT_RANGE |
           others;
CHOICE_S ::= choice_s;                           3.7.3.A    43
COMP ::= pragma;                                 3.7.B      41
COMP ::= var |                                   3.7.B      41
         variant_part |
         null_comp;
COMPILATION ::= compilation;                     10.1.A     69
COMP_ASSOC ::= named |                           4.3.B      47
               EXP;
COMP_REP ::= comp_rep;                           13.4.B     75
COMP_REP ::= pragma;                             13.4.B     75
COMP_REP_S ::= comp_rep_s;                       13.4.B     75
COMP_REP_VOID ::= COMP_REP |                     3.7.B      41
                  void;
COMP_UNIT ::= comp_unit;                         10.1.B     69
COND_CLAUSE ::= cond_clause;                     5.3.A      53
CONSTRAINED ::= constrained;                     3.3.2.B    36
CONSTRAINT ::= RANGE |                           3.3.2.C    37
               float |
               fixed |
               dscrt_range_s |
               dscrmt_aggregate;
CONSTRAINT ::= void;                             3.3.2.B    36
CONTEXT ::= context;                             10.1.1.A   69
CONTEXT_ELEM ::= pragma;                         10.1.B     69
CONTEXT_ELEM ::= use;                            10.1.1.A   69
CONTEXT_ELEM ::= with;                           10.1.1.B   70
DECL ::= REP |                                   3.9.A      44
         use;
DECL ::= constant |                              3.1        33
         var |
         number |
         type |
         subtype |
         subprogram_decl |
         package_decl |
         task_decl |
         generic |
         exception |
         deferred_constant;
DECL ::= pragma;                                 3.1        33
```

```
DECL_S ::= decl_s;                                      7.1.B       62
DEF_CHAR ::= def_char;                                  3.5.1.B     38
DEF_ID ::= attr_id |                                    App. I      76
           pragma_id |
           ARGUMENT;
DEF_ID ::= comp_id;                                     3.7.B       41
DEF_ID ::= const_id;                                    3.2.A       34
DEF_ID ::= dscrmt_id;                                   3.7.1       42
DEF_ID ::= entry_id;                                    9.5.A       66
DEF_ID ::= enum_id;                                     3.5.1.B     38
DEF_ID ::= exception_id;                                11.1        70
DEF_ID ::= function_id;                                 6.1.A       57
DEF_ID ::= generic_id;                                  12.1.A      71
DEF_ID ::= in_id;                                       6.1.C       59
DEF_ID ::= in_out_id |                                  6.1.C       59
           out_id;
DEF_ID ::= iteration_id;                                5.5.B       55
DEF_ID ::= label_id;                                    5.1.B       51
DEF_ID ::= named_stm_id;                                5.5.A       54
DEF_ID ::= number_id;                                   3.2.B       35
DEF_ID ::= package_id;                                  7.1.A       62
DEF_ID ::= private_type_id |                            7.4.A       63
           l_private_type_id;
DEF_ID ::= proc_id;                                     6.1.A       57
DEF_ID ::= subtype_id;                                  3.3.2.A     36
DEF_ID ::= task_body_id;                                9.1.B       65
DEF_ID ::= type_id;                                     3.3.1.A     35
DEF_ID ::= var_id;                                      3.2.A       34
DEF_OCCURRENCE ::= DEF_ID |                             2.3         32
                   DEF_OP |
                   DEF_CHAR;
DEF_OP ::= def_op;                                      6.1.A       57
DESIGNATOR ::= ID |                                     2.3         32
               OP;
DESIGNATOR_CHAR ::= DESIGNATOR |                        4.1.3       46
                    used_char;
DSCRMT_VAR ::= dscrmt_var;                              3.7.1       42
DSCRMT_VAR_S ::= dscrmt_var_s;                          3.7.1       42
DSCRT_RANGE ::= constrained |                           3.6.C       40
                RANGE;
DSCRT_RANGE ::= index;                                  3.6.B       40
DSCRT_RANGE_S ::= dscrt_range_s;                        3.6.A       40
DSCRT_RANGE_VOID ::= DSCRT_RANGE |                      9.5.A       66
                     void;
ENUM_LITERAL ::= enum_id |                              3.5.1.B     38
                 def_char;
EXCEPTION_DEF ::= rename;                               8.5         64
EXCEPTION_DEF ::= void;                                 11.1        70
EXP ::= NAME |                                          4.4.D       49
        numeric_literal |
        null_access |
        aggregate |
        string_literal |
        allocator |
        conversion |
        qualified |
        parenthesized;
EXP ::= aggregate;                                      4.3.A       47
EXP ::= binary;                                         4.4.A       48
EXP ::= membership;                                     4.4.B       48
EXP_CONSTRAINED ::= EXP |                               4.8         50
                    CONSTRAINED;
EXP_S ::= exp_s;                                        4.1.1       46
EXP_VOID ::= EXP |                                      3.2.A       34
             void;
FORMAL_SUBPROG_DEF ::= NAME |                           12.1.C      72
                       box |
                       no_default;
FORMAL_TYPE_SPEC ::= formal_dscrt |                     12.1.D      72
                     formal_integer |
                     formal_fixed |
                     formal_float;
GENERIC_ASSOC ::= ACTUAL;                               12.3.C      73
GENERIC_ASSOC ::= assoc;                                12.3.B      73
```

GENERIC_ASSOC_S ::= generic_assoc_s;	12.3.A	73
GENERIC_HEADER ::= procedure |	12.1.A	71
function |		
package_spec;		
GENERIC_PARAM ::= in |	12.1.C	72
in_out |		
type |		
subprogram_decl;		
GENERIC_PARAM_S ::= generic_param_s;	12.1.B	72
HEADER ::= entry;	9.5.A	66
HEADER ::= function;	6.1.B	58
HEADER ::= procedure;	6.1.B	58
ID ::= DEF_ID |	2.3	32
USED_ID;		
ID_S ::= id_s;	3.2.C	35
INNER_RECORD ::= inner_record;	3.7.3.A	43
ITEM ::= DECL |	3.9.B	44
subprogram_body |		
package_body |		
task_body;		
ITEM_S ::= item_s;	3.9.B	44
ITERATION ::= for |	5.5.B	55
reverse;		
ITERATION ::= void;	5.5.A	54
ITERATION ::= while;	5.5.B	55
LANGUAGE ::= argument_id;	6.1.A	57
LOCATION ::= EXP_VOID |	6.1.A	57
pragma_id;		
LOOP ::= loop;	5.5.A	54
MEMBERSHIP_OP ::= in_op |	4.4.B	48
not_in;		
NAME ::= DESIGNATOR |	4.1.A	45
used_char |		
indexed |		
slice |		
selected |		
all |		
attribute |		
attribute_call;		
NAME ::= function_call;	4.1.B	45
NAME_S ::= name_s;	9.10	68
NAME_VOID ::= NAME |	5.7	56
void;		
OBJECT_DEF ::= EXP_VOID;	3.2.A	34
OBJECT_DEF ::= rename;	8.5	64
OP ::= DEF_OP |	2.3	32
USED_OP;		
PACKAGE_DEF ::= instantiation;	12.3.A	73
PACKAGE_DEF ::= package_spec;	7.1.B	62
PACKAGE_DEF ::= rename;	8.5	64
PACKAGE_SPEC ::= package_spec;	7.1.B	62
PACK_BODY_DESC ::= block |	7.1.A	62
stub |		
rename |		
instantiation |		
void;		
PARAM ::= in;	6.1.C	59
PARAM ::= in_out;	6.1.C	59
PARAM ::= out;	6.1.C	59
PARAM_ASSOC ::= EXP |	6.4	61
assoc;		
PARAM_ASSOC_S ::= param_assoc_s;	2.8.A	33
PARAM_S ::= param_s;	6.1.C	59
PRAGMA ::= pragma;	2.8.A	33
PRAGMA_S ::= pragma_s;	10.1.B	69
RANGE ::= range |	3.5	37
attribute |		
attribute_call;		
RANGE_VOID ::= RANGE |	3.5.7	39
void;		
REP ::= simple_rep |	13.1	74
address |		
record_rep;		
REP_VOID ::= REP |	3.7.A	41

```
                void;
SELECT_CLAUSE ::= pragma;                                9.7.1.B   67
SELECT_CLAUSE ::= select_clause;                         9.7.1.B   67
SELECT_CLAUSE_S ::= select_clause_s;                     9.7.1.A   67
SHORT_CIRCUIT_OP ::= and_then |                          4.4.A     48
                     or_else;
STM ::= if |                                             5.1.D     52
        case |
        named_stm |
        LOOP |
        block |
        accept |
        select |
        cond_entry |
        timed_entry;
STM ::= labeled;                                         5.1.B     51
STM ::= null_stm |                                       5.1.C     51
        assign |
        procedure_call |
        exit |
        return |
        goto |
        entry_call |
        delay |
        abort |
        raise |
        code;
STM ::= pragma;                                          5.1.C     51
STM ::= terminate;                                       9.7.1.B   67
STM_S ::= stm_s;                                         5.1.A     51
SUBPROGRAM_DEF ::= FORMAL_SUBPROG_DEF;                   12.1.C    72
SUBPROGRAM_DEF ::= instantiation;                        12.3.A    73
SUBPROGRAM_DEF ::= rename;                               8.5       64
SUBPROGRAM_DEF ::= void;                                 6.1.A     57
SUBP_BODY_DESC ::= block |                               6.1.A     57
                   stub |
                   instantiation |
                   FORMAL_SUBPROG_DEF |
                   rename |
                   LANGUAGE |
                   void;
SUBUNIT_BODY ::= subprogram_body |                       10.2.A    70
                 package_body |
                 task_body;
TASK_DEF ::= task_spec;                                  9.1.A     65
TYPE_RANGE ::= RANGE |                                   4.4.B     48
               NAME;
TYPE_SPEC ::= CONSTRAINED;                               3.2.A     34
TYPE_SPEC ::= FORMAL_TYPE_SPEC;                          12.1.D    72
TYPE_SPEC ::= enum_literal_s |                           3.3.1.B   36
              integer |
              fixed |
              float |
              array |
              record |
              access |
              derived;
TYPE_SPEC ::= l_private;                                 7.4.A     63
TYPE_SPEC ::= private;                                   7.4.A     63
TYPE_SPEC ::= task_spec;                                 9.1.A     65
TYPE_SPEC ::= universal_integer |                        App. I    76
              universal_fixed |
              universal_real;
TYPE_SPEC ::= void;                                      3.8.1     44
UNIT_BODY ::= package_body |                             10.1.B    69
              package_decl |
              subunit |
              generic |
              subprogram_body |
              subprogram_decl |
              void;
USED_ID ::= used_object_id |                             4.1.A     45
            used_name_id |
            used_bltn_id;
```

```
USED_OP ::= used_op |                                 4.1.A      45
            used_bltn_op;
VARIANT ::= variant;                                  3.7.3.A    43
VARIANT_S ::= variant_s;                              3.7.3.A    43
abort => as_name_s:NAME_S;                            9.10       68
abort => lx_srcpos:source_position,                   9.10       68
         lx_comments:comments;
accept => as_name:NAME,                               9.5.C      66
          as_param_s:PARAM_S,
          as_stm_s:STM_S;
accept => lx_srcpos:source_position,                  9.5.C      66
          lx_comments:comments;
access => as_constrained:CONSTRAINED;                 3.8        44
access => lx_srcpos:source_position,                  3.8        44
          lx_comments:comments;
access => sm_size:EXP_VOID,                           3.8        44
          sm_storage_size:EXP_VOID,
          sm_controlled:Boolean;
address => as_name:NAME,                              13.5       75
           as_exp:EXP;
address => lx_srcpos:source_position,                 13.5       75
           lx_comments:comments;
aggregate => as_list:seq of COMP_ASSOC;               4.3.A      47
aggregate => lx_srcpos:source_position,               4.3.A      47
             lx_comments:comments;
aggregate => sm_exp_type:TYPE_SPEC,                   4.3.A      47
             sm_constraint:CONSTRAINT,
             sm_normalized_comp_s:EXP_S;
alignment => as_pragma_s:PRAGMA_S,                    13.4.A     74
             as_exp_void:EXP_VOID;
all => as_name:NAME;                                  4.1.3      46
all => lx_srcpos:source_position,                     4.1.3      46
       lx_comments:comments;
all => sm_exp_type:TYPE_SPEC;                         4.1.3      46
allocator => as_exp_constrained:EXP_CONSTRAINED;      4.8        50
allocator => lx_srcpos:source_position,               4.8        50
             lx_comments:comments;
allocator => sm_exp_type:TYPE_SPEC,                   4.8        50
             sm_value:value;
alternative => as_choice_s:CHOICE_S,                  5.4        53
               as_stm_s:STM_S;
alternative => lx_srcpos:source_position,             5.4        53
               lx_comments:comments;
alternative_s => as_list:seq of ALTERNATIVE;          5.4        53
alternative_s => lx_srcpos:source_position,           5.4        53
                 lx_comments:comments;
and_then => lx_srcpos:source_position,                4.4.A      48
            lx_comments:comments;
argument_id => lx_symrep:symbol_rep;                  App. I     76
array => as_dscrt_range_s:DSCRT_RANGE_S,              3.6.A      40
         as_constrained:CONSTRAINED;
array => lx_srcpos:source_position,                   3.6.A      40
         lx_comments:comments;
array => sm_size:EXP_VOID,                            3.6.A      40
         sm_packing:Boolean;
assign => as_name:NAME,                               5.2        52
          as_exp:EXP;
assign => lx_srcpos:source_position,                  5.2        52
          lx_comments:comments;
assoc => as_designator:DESIGNATOR,                    6.4        61
         as_actual:ACTUAL;
assoc => lx_srcpos:source_position,                   6.4        61
         lx_comments:comments;
attr_id => lx_symrep:symbol_rep;                      App. I     76
attribute => as_name:NAME,                            4.1.4      47
             as_id:ID;
attribute => lx_srcpos:source_position,               4.1.4      47
             lx_comments:comments;
attribute => sm_exp_type:TYPE_SPEC,                   4.1.4      47
             sm_value:value;
attribute_call => as_name:NAME,                       4.1.4      47
                  as_exp:EXP;
attribute_call => lx_srcpos:source_position,          4.1.4      47
                  lx_comments:comments;
```

attribute_call =>	sm_exp_type: TYPE_SPEC, sm_value: value;	4.1.4	47
binary =>	as_exp1: EXP, as_binary_op: BINARY_OP, as_exp2: EXP;	4.4.A	48
binary =>	lx_srcpos: source_position, lx_comments: comments;	4.4.A	48
binary =>	sm_exp_type: TYPE_SPEC, sm_value: value;	4.4.A	48
block =>	as_item_s: ITEM_S, as_stm_s: STM_S, as_alternative_s: ALTERNATIVE_S;	5.6	55
block =>	lx_srcpos: source_position, lx_comments: comments;	5.6	55
box =>	lx_srcpos: source_position, lx_comments: comments;	12.1.C	72
case =>	as_exp: EXP, as_alternative_s: ALTERNATIVE_S;	5.4	53
case =>	lx_srcpos: source_position, lx_comments: comments;	5.4	53
choice_s =>	as_list: seq of CHOICE;	3.7.3.A	43
choice_s =>	lx_srcpos: source_position, lx_comments: comments;	3.7.3.A	43
code =>	as_name: NAME, as_exp: EXP;	13.8	75
code =>	lx_srcpos: source_position, lx_comments: comments;	13.8	75
comp_id =>	lx_srcpos: source_position, lx_comments: comments, lx_symrep: symbol_rep;	3.7.B	41
comp_id =>	sm_obj_type: TYPE_SPEC, sm_init_exp: EXP_VOID, sm_comp_spec: COMP_REP_VOID;	3.7.B	41
comp_rep =>	as_name: NAME, as_exp: EXP, as_range: RANGE;	13.4.B	75
comp_rep =>	lx_srcpos: source_position, lx_comments: comments;	13.4.B	75
comp_rep_s =>	as_list: seq of COMP_REP;	13.4.B	75
comp_rep_s =>	lx_srcpos: source_position, lx_comments: comments;	13.4.B	75
comp_unit =>	as_context: CONTEXT, as_unit_body: UNIT_BODY, as_pragma_s: PRAGMA_S;	10.1.B	69
comp_unit =>	lx_srcpos: source_position, lx_comments: comments;	10.1.B	69
compilation =>	as_list: seq of COMP_UNIT;	10.1.A	69
compilation =>	lx_srcpos: source_position, lx_comments: comments;	10.1.A	69
cond_clause =>	as_exp_void: EXP_VOID, as_stm_s: STM_S;	5.3.A	53
cond_clause =>	lx_srcpos: source_position, lx_comments: comments;	5.3.A	53
cond_entry =>	as_stm_s1: STM_S, as_stm_s2: STM_S;	9.7.2	68
cond_entry =>	lx_srcpos: source_position, lx_comments: comments;	9.7.2	68
const_id =>	lx_srcpos: source_position, lx_comments: comments, lx_symrep: symbol_rep;	3.2.A	34
const_id =>	sm_address: EXP_VOID, sm_obj_type: TYPE_SPEC, sm_obj_def: OBJECT_DEF, sm_first: DEF_OCCURRENCE;	3.2.A	34
constant =>	as_id_s: ID_S, as_type_spec: TYPE_SPEC, as_object_def: OBJECT_DEF;	3.2.A	34
constant =>	lx_srcpos: source_position, lx_comments: comments;	3.2.A	34
constrained =>	as_name: NAME, as_constraint: CONSTRAINT;	3.3.2.B	36
constrained =>	cd_impl_size: integer;	3.3.2.B	36
constrained =>	lx_srcpos: source_position, lx_comments: comments;	3.3.2.B	36

```
constrained => sm_type_struct:TYPE_SPEC,                3.3.2.B   36
               sm_base_type:TYPE_SPEC,
               sm_constraint:CONSTRAINT;
context => as_list:seq of CONTEXT_ELEM;                 10.1.1.A  69
context => lx_srcpos:source_position,                   10.1.1.A  69
           lx_comments:comments;
conversion => as_name:NAME,                             4.6       50
              as_exp:EXP;
conversion => lx_srcpos:source_position,                4.6       50
              lx_comments:comments;
conversion => sm_exp_type:TYPE_SPEC,                    4.6       50
              sm_value:value;
decl_s => as_list:seq of DECL;                          7.1.B     62
decl_s => lx_srcpos:source_position,                    7.1.B     62
          lx_comments:comments;
def_char => lx_srcpos:source_position,                  3.5.1.B   38
            lx_comments:comments,
            lx_symrep:symbol_rep;
def_char => sm_obj_type:TYPE_SPEC,                      3.5.1.B   38
            sm_pos:integer,
            sm_rep:integer;
def_op => lx_srcpos:source_position,                    6.1.A     57
          lx_comments:comments,
          lx_symrep:symbol_rep;
def_op => sm_spec:HEADER,                               6.1.A     57
          sm_body:SUBP_BODY_DESC,
          sm_location:LOCATION,
          sm_stub:DEF_OCCURRENCE,
          sm_first:DEF_OCCURRENCE;
deferred_constant => as_id_s:ID_S,                      7.4.B     63
                     as_name:NAME;
deferred_constant => lx_srcpos:source_position,         7.4.B     63
                     lx_comments:comments;
delay => as_exp:EXP;                                    9.6       66
delay => lx_srcpos:source_position,                     9.6       66
         lx_comments:comments;
derived => as_constrained:CONSTRAINED;                  3.4       37
derived => cd_impl_size:integer;                        3.4       37
derived => lx_srcpos:source_position,                   3.4       37
           lx_comments:comments;
derived => sm_size:EXP_VOID,                            3.4       37
           sm_actual_delta:Rational,
           sm_packing:Boolean,
           sm_controlled:Boolean,
           sm_storage_size:EXP_VOID;
dscrmt_aggregate => as_list:seq of COMP_ASSOC;          3.7.2     42
dscrmt_aggregate => lx_srcpos:source_position,          3.7.2     42
                    lx_comments:comments;
dscrmt_aggregate => sm_normalized_comp_s:EXP_S;         3.7.2     42
dscrmt_id => lx_srcpos:source_position,                 3.7.1     42
             lx_comments:comments,
             lx_symrep:symbol_rep;
dscrmt_id => sm_obj_type:TYPE_SPEC,                     3.7.1     42
             sm_init_exp:EXP_VOID,
             sm_first:DEF_OCCURRENCE,
             sm_comp_spec:COMP_REP_VOID;
dscrmt_var => as_id_s:ID_S,                             3.7.1     42
              as_name:NAME,
              as_object_def:OBJECT_DEF;
dscrmt_var => lx_srcpos:source_position,                3.7.1     42
              lx_comments:comments;
dscrmt_var_s => as_list:seq of DSCRMT_VAR;              3.7.1     42
dscrmt_var_s => lx_srcpos:source_position,              3.7.1     42
                lx_comments:comments;
dscrt_range_s => as_list:seq of DSCRT_RANGE;            3.6.A     40
dscrt_range_s => lx_srcpos:source_position,             3.6.A     40
                 lx_comments:comments;
entry => as_dscrt_range_void:DSCRT_RANGE_VOID,          9.5.A     66
         as_param_s:PARAM_S;
entry => lx_srcpos:source_position,                     9.5.A     66
         lx_comments:comments;
entry_call => as_name:NAME,                             9.5.B     66
              as_param_assoc_s:PARAM_ASSOC_S;
entry_call => lx_srcpos:source_position,                9.5.B     66
```

```
                 lx_comments:comments;
entry_call => sm_normalized_param_s:EXP_S;           9.5.B    66
entry_id => lx_srcpos:source_position,               9.5.A    66
            lx_comments:comments,
            lx_symrep:symbol_rep;
entry_id => sm_spec:HEADER,                          9.5.A    66
            sm_address:EXP_VOID;
enum_id => lx_srcpos:source_position,                3.5.1.B  38
           lx_comments:comments,
           lx_symrep:symbol_rep;
enum_id => sm_obj_type:TYPE_SPEC,                    3.5.1.B  38
           sm_pos:integer,
           sm_rep:integer;
enum_literal_s => as_list:seq of ENUM_LITERAL;       3.5.1.A  37
enum_literal_s => cd_impl_size:integer;              3.5.1.A  37
enum_literal_s => lx_srcpos:source_position,         3.5.1.A  37
                  lx_comments:comments;
enum_literal_s => sm_size:EXP_VOID;                  3.5.1.A  37
exception => as_id_s:ID_S,                           11.1     70
             as_exception_def:EXCEPTION_DEF;
exception => lx_srcpos:source_position,              11.1     70
             lx_comments:comments;
exception_id => lx_srcpos:source_position,           11.1     70
                lx_comments:comments,
                lx_symrep:symbol_rep;
exception_id => sm_exception_def:EXCEPTION_DEF;      11.1     70
exit => as_name_void:NAME_VOID,                      5.7      56
        as_exp_void:EXP_VOID;
exit => lx_srcpos:source_position,                   5.7      56
        lx_comments:comments;
exit => sm_stm:LOOP;                                 5.7      56
exp_s => as_list:seq of EXP;                         4.1.1    46
exp_s => lx_srcpos:source_position,                  4.1.1    46
         lx_comments:comments;
fixed => as_exp:EXP,                                 3.5.9    39
         as_range_void:RANGE_VOID;
fixed => cd_impl_size:integer;                       3.5.9    39
fixed => lx_srcpos:source_position,                  3.5.9    39
         lx_comments:comments;
fixed => sm_size:EXP_VOID,                           3.5.9    39
         sm_actual_delta:Rational,
         sm_bits:integer,
         sm_base_type:TYPE_SPEC;
float => as_exp:EXP,                                 3.5.7    39
         as_range_void:RANGE_VOID;
float => cd_impl_size:integer;                       3.5.7    39
float => lx_srcpos:source_position,                  3.5.7    39
         lx_comments:comments;
float => sm_size:EXP_VOID,                           3.5.7    39
         sm_type_struct:TYPE_SPEC,
         sm_base_type:TYPE_SPEC;
for => as_id:ID,                                     5.5.B    55
       as_dscrt_range:DSCRT_RANGE;
for => lx_srcpos:source_position,                    5.5.B    55
       lx_comments:comments;
formal_dscrt => lx_srcpos:source_position,           12.1.D   72
                lx_comments:comments;
formal_fixed => lx_srcpos:source_position,           12.1.D   72
                lx_comments:comments;
formal_float => lx_srcpos:source_position,           12.1.D   72
                lx_comments:comments;
formal_integer => lx_srcpos:source_position,         12.1.D   72
                  lx_comments:comments;
function => as_param_s:PARAM_S,                      6.1.B    58
            as_name_void:NAME_VOID;
function => lx_srcpos:source_position,               6.1.B    58
            lx_comments:comments;
function_call => as_name:NAME,                       6.4      61
                 as_param_assoc_s:PARAM_ASSOC_S;
function_call => lx_srcpos:source_position,          6.4      61
                 lx_comments:comments;
function_call => sm_exp_type:TYPE_SPEC,              6.4      61
                 sm_value:value,
                 sm_normalized_param_s:EXP_S,
```

```
                  lx_prefix: Boolean;
function_id => lx_srcpos: source_position,                  6.1.A     57
               lx_comments: comments,
               lx_symrep: symbol_rep;
function_id => sm_spec: HEADER,                             6.1.A     57
               sm_body: SUBP_BODY_DESC,
               sm_location: LOCATION,
               sm_stub: DEF_OCCURRENCE,
               sm_first: DEF_OCCURRENCE;
generic => as_id: ID,                                       12.1.A    71
           as_generic_param_s: GENERIC_PARAM_S,
           as_generic_header: GENERIC_HEADER;
generic => lx_srcpos: source_position,                      12.1.A    71
           lx_comments: comments;
generic_assoc_s => as_list: seq of GENERIC_ASSOC;           12.3.A    73
generic_assoc_s => lx_srcpos: source_position,              12.3.A    73
                   lx_comments: comments;
generic_id => lx_symrep: symbol_rep,                        12.1.A    71
              lx_srcpos: source_position,
              lx_comments: comments;
generic_id => sm_generic_param_s: GENERIC_PARAM_S,          12.1.A    71
              sm_spec: GENERIC_HEADER,
              sm_body: BLOCK_STUB_VOID,
              sm_first: DEF_OCCURRENCE,
              sm_stub: DEF_OCCURRENCE;
generic_param_s => as_list: seq of GENERIC_PARAM;           12.1.B    72
generic_param_s => lx_srcpos: source_position,              12.1.B    72
                   lx_comments: comments;
goto => as_name: NAME;                                      5.9       56
goto => lx_srcpos: source_position,                         5.9       56
        lx_comments: comments;
id_s => as_list: seq of ID;                                 3.2.C     35
id_s => lx_srcpos: source_position,                         3.2.C     35
        lx_comments: comments;
if => as_list: seq of COND_CLAUSE;                          5.3.A     53
if => lx_srcpos: source_position,                           5.3.A     53
      lx_comments: comments;
in => as_id_s: ID_S,                                        6.1.C     59
      as_name: NAME,
      as_exp_void: EXP_VOID;
in => lx_srcpos: source_position,                           6.1.C     59
      lx_comments: comments,
      lx_default: Boolean;
in_id => lx_srcpos: source_position,                        6.1.C     59
         lx_comments: comments,
         lx_symrep: symbol_rep;
in_id => sm_obj_type: TYPE_SPEC,                            6.1.C     59
         sm_init_exp: EXP_VOID,
         sm_first: DEF_OCCURRENCE;
in_op => lx_srcpos: source_position,                        4.4.B     48
         lx_comments: comments;
in_out => as_id_s: ID_S,                                    6.1.C     59
          as_name: NAME,
          as_exp_void: EXP_VOID;
in_out => lx_srcpos: source_position,                       6.1.C     59
          lx_comments: comments;
in_out_id => lx_srcpos: source_position,                    6.1.C     59
             lx_comments: comments,
             lx_symrep: symbol_rep;
in_out_id => sm_obj_type: TYPE_SPEC,                        6.1.C     59
             sm_first: DEF_OCCURRENCE;
index => as_name: NAME;                                     3.6.B     40
index => lx_srcpos: source_position,                        3.6.B     40
         lx_comments: comments;
indexed => as_name: NAME,                                   4.1.1     46
           as_exp_s: EXP_S;
indexed => lx_srcpos: source_position,                      4.1.1     46
           lx_comments: comments;
indexed => sm_exp_type: TYPE_SPEC;                          4.1.1     46
inner_record => as_list: seq of COMP;                       3.7.3.A   43
inner_record => lx_srcpos: source_position,                 3.7.3.A   43
                lx_comments: comments;
instantiation => as_name: NAME,                             12.3.A    73
                 as_generic_assoc_s: GENERIC_ASSOC_S;
```

```
instantiation => lx_srcpos:source_position,            12.3.A    73
                 lx_comments:comments;
instantiation => sm_decl_s:DECL_S;                      12.3.A    73
integer => as_range:RANGE;                              3.5.4     38
integer => cd_impl_size:integer;                        3.5.4     38
integer => lx_srcpos:source_position,                   3.5.4     38
           lx_comments:comments;
integer => sm_size:EXP_VOID,                            3.5.4     38
           sm_type_struct:TYPE_SPEC,
           sm_base_type:TYPE_SPEC;
item_s => as_list:seq of ITEM;                          3.9.B     44
item_s => lx_srcpos:source_position,                    3.9.B     44
          lx_comments:comments;
iteration_id => lx_srcpos:source_position,              5.5.B     55
                lx_comments:comments,
                lx_symrep:symbol_rep;
iteration_id => sm_obj_type:TYPE_SPEC;                  5.5.B     55
l_private => lx_srcpos:source_position,                 7.4.A     63
             lx_comments:comments;
l_private => sm_discriminants:DSCRMT_VAR_S;             7.4.A     63
l_private_type_id => lx_srcpos:source_position,         7.4.A     63
                     lx_comments:comments,
                     lx_symrep:symbol_rep;
l_private_type_id => sm_type_spec:TYPE_SPEC;            7.4.A     63
label_id => lx_srcpos:source_position,                  5.1.B     51
            lx_comments:comments,
            lx_symrep:symbol_rep;
label_id => sm_stm:STM;                                 5.1.B     51
labeled => as_id_s:ID_S,                                5.1.B     51
           as_stm:STM;
labeled => lx_srcpos:source_position,                   5.1.B     51
           lx_comments:comments;
loop => as_iteration:ITERATION,                         5.5.A     54
        as_stm_s:STM_S;
loop => lx_srcpos:source_position,                      5.5.A     54
        lx_comments:comments;
membership => as_exp:EXP,                               4.4.B     48
              as_membership_op:MEMBERSHIP_OP,
              as_type_range:TYPE_RANGE;
membership => lx_srcpos:source_position,                4.4.B     48
              lx_comments:comments;
membership => sm_exp_type:TYPE_SPEC,                    4.4.B     48
              sm_value:value;
name_s => as_list:seq of NAME;                          9.10      68
name_s => lx_srcpos:source_position,                    9.10      68
          lx_comments:comments;
named => as_choice_s:CHOICE_S,                          4.3.B     47
         as_exp:EXP;
named => lx_srcpos:source_position,                     4.3.B     47
         lx_comments:comments;
named_stm => as_id:ID,                                  5.5.A     54
             as_stm:STM;
named_stm => lx_srcpos:source_position,                 5.5.A     54
             lx_comments:comments;
named_stm_id => lx_srcpos:source_position,              5.5.A     54
                lx_comments:comments,
                lx_symrep:symbol_rep;
named_stm_id => sm_stm:STM;                             5.5.A     54
no_default => lx_srcpos:source_position,                12.1.C    72
              lx_comments:comments;
not_in => lx_srcpos:source_position,                    4.4.B     48
          lx_comments:comments;
null_access => lx_srcpos:source_position,               4.4.D     49
               lx_comments:comments;
null_access => sm_exp_type:TYPE_SPEC,                   4.4.D     49
               sm_value:value;
null_comp => lx_srcpos:source_position,                 3.7.B     41
             lx_comments:comments;
null_stm => lx_srcpos:source_position,                  5.1.F     52
            lx_comments:comments;
number => as_id_s:ID_S,                                 3.2.B     35
          as_exp:EXP;
number => lx_srcpos:source_position,                    3.2.B     35
          lx_comments:comments;
```

number_id => lx_srcpos:source_position,	3.2.B	35
lx_comments:comments,		
lx_symrep:symbol_rep;		
number_id => sm_obj_type:TYPE_SPEC,	3.2.B	35
sm_init_exp:EXP;		
numeric_literal => lx_srcpos:source_position,	4.4.D	49
lx_comments:comments,		
lx_numrep:number_rep;		
numeric_literal => sm_exp_type:TYPE_SPEC,	4.4.D	49
sm_value:value;		
or_else => lx_srcpos:source_position,	4.4.A	48
lx_comments:comments;		
others => lx_srcpos:source_position,	3.7.3.B	43
lx_comments:comments;		
out => as_id_s:ID_S,	6.1.C	59
as_name:NAME,		
as_exp_void:EXP_VOID;		
out => lx_srcpos:source_position,	6.1.C	59
lx_comments:comments;		
out_id => lx_srcpos:source_position,	6.1.C	59
lx_comments:comments,		
lx_symrep:symbol_rep;		
out_id => sm_obj_type:TYPE_SPEC,	6.1.C	59
sm_first:DEF_OCCURRENCE;		
package_body => as_id:ID,	7.1.C	63
as_block_stub:BLOCK_STUB;		
package_body => lx_srcpos:source_position,	7.1.C	63
lx_comments:comments;		
package_decl => as_id:ID,	7.1.A	62
as_package_def:PACKAGE_DEF;		
package_decl => lx_srcpos:source_position,	7.1.A	62
lx_comments:comments;		
package_id => lx_srcpos:source_position,	7.1.A	62
lx_comments:comments,		
lx_symrep:symbol_rep;		
package_id => sm_spec:PACKAGE_SPEC,	7.1.A	62
sm_body:PACK_BODY_DESC,		
sm_address:EXP_VOID,		
sm_stub:DEF_OCCURRENCE,		
sm_first:DEF_OCCURRENCE;		
package_spec => as_decl_s1:DECL_S,	7.1.B	62
as_decl_s2:DECL_S;		
package_spec => lx_srcpos:source_position,	7.1.B	62
lx_comments:comments;		
param_assoc_s => as_list:seq of PARAM_ASSOC;	2.8.A	33
param_assoc_s => lx_srcpos:source_position,	2.8.A	33
lx_comments:comments;		
param_s => as_list:seq of PARAM;	6.1.C	59
param_s => lx_srcpos:source_position,	6.1.C	59
lx_comments:comments;		
parenthesized => as_exp:EXP;	4.4.D	49
parenthesized => lx_srcpos:source_position,	4.4.D	49
lx_comments:comments;		
parenthesized => sm_exp_type:TYPE_SPEC,	4.4.D	49
sm_value:value;		
pragma => as_id:ID,	2.8.A	33
as_param_assoc_s:PARAM_ASSOC_S;		
pragma => lx_srcpos:source_position,	2.8.A	33
lx_comments:comments;		
pragma_id => as_list:seq of ARGUMENT;	App. I	76
pragma_id => lx_symrep:symbol_rep;	App. I	76
pragma_s => as_list:seq of PRAGMA;	10.1.B	69
pragma_s => lx_srcpos:source_position,	10.1.B	69
lx_comments:comments;		
private => lx_srcpos:source_position,	7.4.A	63
lx_comments:comments;		
private => sm_discriminants:DSCRMT_VAR_S;	7.4.A	63
private_type_id => lx_srcpos:source_position,	7.4.A	63
lx_comments:comments,		
lx_symrep:symbol_rep;		
private_type_id => sm_type_spec:TYPE_SPEC;	7.4.A	63
proc_id => lx_srcpos:source_position,	6.1.A	57
lx_comments:comments,		
lx_symrep:symbol_rep;		

proc_id => sm_spec:HEADER,	6.1.A	57
sm_body:SUBP_BODY_DESC,		
sm_location:LOCATION,		
sm_stub:DEF_OCCURRENCE,		
sm_first:DEF_OCCURRENCE;		
procedure => as_param_s:PARAM_S;	6.1.B	58
procedure => lx_srcpos:source_position,	6.1.B	58
lx_comments:comments;		
procedure_call => as_name:NAME,	6.4	61
as_param_assoc_s:PARAM_ASSOC_S;		
procedure_call => lx_srcpos:source_position,	6.4	61
lx_comments:comments;		
procedure_call => sm_normalized_param_s:EXP_S;	6.4	61
qualified => as_name:NAME,	4.7	50
as_exp:EXP;		
qualified => lx_srcpos:source_position,	4.7	50
lx_comments:comments;		
qualified => sm_exp_type:TYPE_SPEC,	4.7	50
sm_value:value;		
raise => as_name_void:NAME_VOID;	11.3	71
raise => lx_srcpos:source_position,	11.3	71
lx_comments:comments;		
range => as_exp1:EXP,	3.5	37
as_exp2:EXP;		
range => lx_srcpos:source_position,	3.5	37
lx_comments:comments;		
range => sm_base_type:TYPE_SPEC;	3.5	37
record => as_list:seq of COMP;	3.7.A	41
record => lx_srcpos:source_position,	3.7.A	41
lx_comments:comments;		
record => sm_size:EXP_VOID,	3.7.A	41
sm_discriminants:DSCRMT_VAR_S,		
sm_packing:Boolean,		
sm_record_spec:REP_VOID;		
record_rep => as_name:NAME,	13.4.A	74
as_alignment:ALIGNMENT,		
as_comp_rep_s:COMP_REP_S;		
record_rep => lx_srcpos:source_position,	13.4.A	74
lx_comments:comments;		
rename => as_name:NAME;	8.5	64
rename => lx_srcpos:source_position,	8.5	64
lx_comments:comments;		
return => as_exp_void:EXP_VOID;	5.8	56
return => lx_srcpos:source_position,	5.8	56
lx_comments:comments;		
reverse => as_id:ID,	5.5.B	55
as_dscrt_range:DSCRT_RANGE;		
reverse => lx_srcpos:source_position,	5.5.B	55
lx_comments:comments;		
select => as_select_clause_s:SELECT_CLAUSE_S,	9.7.1.A	67
as_stm_s:STM_S;		
select => lx_srcpos:source_position,	9.7.1.A	67
lx_comments:comments;		
select_clause => as_exp_void:EXP_VOID,	9.7.1.B	67
as_stm_s:STM_S;		
select_clause => lx_srcpos:source_position,	9.7.1.B	67
lx_comments:comments;		
select_clause_s => as_list:seq of SELECT_CLAUSE;	9.7.1.A	67
select_clause_s => lx_srcpos:source_position,	9.7.1.A	67
lx_comments:comments;		
selected => as_name:NAME,	4.1.3	46
as_designator_char:DESIGNATOR_CHAR;		
selected => lx_srcpos:source_position,	4.1.3	46
lx_comments:comments;		
selected => sm_exp_type:TYPE_SPEC;	4.1.3	46
simple_rep => as_name:NAME,	13.3	74
as_exp:EXP;		
simple_rep => lx_srcpos:source_position,	13.3	74
lx_comments:comments;		
slice => as_name:NAME,	4.1.2	46
as_dscrt_range:DSCRT_RANGE;		
slice => lx_srcpos:source_position,	4.1.2	46
lx_comments:comments;		
slice => sm_exp_type:TYPE_SPEC,	4.1.2	46

```
        sm_constraint:CONSTRAINT;
stm_s => as_list:seq of STM;                          5.1.A     51
stm_s => lx_srcpos:source_position,                   5.1.A     51
         lx_comments:comments;
string_literal => lx_srcpos:source_position,          4.4.D     49
                  lx_comments:comments,
                  lx_symrep:symbol_rep;
string_literal => sm_exp_type:TYPE_SPEC,              4.4.D     49
                  sm_constraint:CONSTRAINT,
                  sm_value:value;
stub => lx_srcpos:source_position,                    10.2.B    70
        lx_comments:comments;
subprogram_body => as_designator:DESIGNATOR,          6.3       60
                   as_header:HEADER,
                   as_block_stub:BLOCK_STUB;
subprogram_body => lx_srcpos:source_position,         6.3       60
                   lx_comments:comments;
subprogram_decl => as_designator:DESIGNATOR,          6.1.A     57
                   as_header:HEADER,
                   as_subprogram_def:SUBPROGRAM_DEF;
subprogram_decl => lx_srcpos:source_position,         6.1.A     57
                   lx_comments:comments;
subtype => as_id:ID,                                  3.3.2.A   36
           as_constrained:CONSTRAINED;
subtype => lx_srcpos:source_position,                 3.3.2.A   36
           lx_comments:comments;
subtype_id => lx_srcpos:source_position,              3.3.2.A   36
              lx_comments:comments,
              lx_symrep:symbol_rep;
subtype_id => sm_type_spec:CONSTRAINED;               3.3.2.A   36
subunit => as_name:NAME,                              10.2.A    70
           as_subunit_body:SUBUNIT_BODY;
subunit => lx_srcpos:source_position,                 10.2.A    70
           lx_comments:comments;
task_body => as_id:ID,                                9.1.B     65
             as_block_stub:BLOCK_STUB;
task_body => lx_srcpos:source_position,               9.1.B     65
             lx_comments:comments;
task_body_id => lx_srcpos:source_position,            9.1.B     65
                lx_comments:comments,
                lx_symrep:symbol_rep;
task_body_id => sm_type_spec:TYPE_SPEC,               9.1.B     65
                sm_body:BLOCK_STUB_VOID,
                sm_first:DEF_OCCURRENCE,
                sm_stub:DEF_OCCURRENCE;
task_decl => as_id:ID,                                9.1.A     65
             as_task_def:TASK_DEF;
task_decl => lx_srcpos:source_position,               9.1.A     65
             lx_comments:comments;
task_spec => as_decl_s:DECL_S;                        9.1.A     65
task_spec => lx_srcpos:source_position,               9.1.A     65
             lx_comments:comments;
task_spec => sm_body:BLOCK_STUB_VOID,                 9.1.A     65
             sm_address:EXP_VOID,
             sm_storage_size:EXP_VOID;
terminate => lx_srcpos:source_position,               9.7.1.B   67
             lx_comments:comments;
timed_entry => as_stm_s1:STM_S,                       9.7.3     68
               as_stm_s2:STM_S;
timed_entry => lx_srcpos:source_position,             9.7.3     68
               lx_comments:comments;
type => as_id:ID,                                     3.3.1.A   35
        as_dscrmt_var_s:DSCRMT_VAR_S,
        as_type_spec:TYPE_SPEC;
type => lx_srcpos:source_position,                    3.3.1.A   35
        lx_comments:comments;
type_id => lx_srcpos:source_position,                 3.3.1.A   35
           lx_comments:comments,
           lx_symrep:symbol_rep;
type_id => sm_type_spec:TYPE_SPEC,                    3.3.1.A   35
           sm_first:DEF_OCCURRENCE;
universal_fixed => ;                                  App. I    76
universal_integer => ;                                App. I    76
universal_real => ;                                   App. I    76
```

use => as_list:seq of NAME;	8.4	64
use => lx_srcpos:source_position, lx_comments:comments;	8.4	64
used_bltn_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_bltn_id => sm_operator:operator;	4.1.A	45
used_bltn_op => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_bltn_op => sm_operator:operator;	4.1.A	45
used_char => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_char => sm_defn:DEF_OCCURRENCE, sm_exp_type:TYPE_SPEC, sm_value:value;	4.1.A	45
used_name_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_name_id => sm_defn:DEF_OCCURRENCE;	4.1.A	45
used_object_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_object_id => sm_exp_type:TYPE_SPEC, sm_defn:DEF_OCCURRENCE, sm_value:value;	4.1.A	45
used_op => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	4.1.A	45
used_op => sm_defn:DEF_OCCURRENCE;	4.1.A	45
var => as_id_s:ID_S, as_type_spec:TYPE_SPEC, as_object_def:OBJECT_DEF;	3.2.A	34
var => lx_srcpos:source_position, lx_comments:comments;	3.2.A	34
var_id => lx_srcpos:source_position, lx_comments:comments, lx_symrep:symbol_rep;	3.2.A	34
var_id => sm_obj_type:TYPE_SPEC, sm_address:EXP_VOID, sm_obj_def:OBJECT_DEF;	3.2.A	34
variant => as_choice_s:CHOICE_S, as_record:INNER_RECORD;	3.7.3.A	43
variant => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
variant_part => as_name:NAME, as_variant_s:VARIANT_S;	3.7.3.A	43
variant_part => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
variant_s => as_list:seq of VARIANT;	3.7.3.A	43
variant_s => lx_srcpos:source_position, lx_comments:comments;	3.7.3.A	43
void => ;	2	32
while => as_exp:EXP;	5.5.B	55
while => lx_srcpos:source_position, lx_comments:comments;	5.5.B	55
with => as_list:seq of NAME;	10.1.1.B	70
with => lx_srcpos:source_position, lx_comments:comments;	10.1.1.B	70

APPENDIX V
DIANA NAMES

This appendix is an index of all of the names which occur in the DIANA definition: these names include class names, node names, attribute labels, and attribute types. Each name is shown in the form

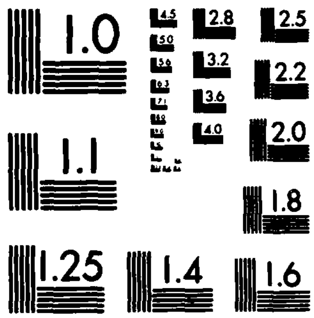
`name [section-number-list] page-number-list`

The section number list gives all the sections of Chapter 2 which make use of the name. The page number list gives pages of this document on which the name may be found. Either list may be split across several lines.

AD-A128 232 DIANA REFERENCE MANUAL REVISION 3(U) TARTAN LABS INC 3/3
PITTSBURGH PA A EVANS ET AL. 28 FEB 83 TL-83-4
MDA903-82-C-0148

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX VI
DIANA ATTRIBUTES

This appendix is an index of all of the attributes which occur in DIANA tree nodes. Each attribute is shown in the form

```
label : type [section-number-list] page-number-list
```

The section number list gives all the sections of Chapter 2 which make use of the attribute. The page number list gives pages of this document on which the attribute may be found. Either list may be split across several lines. The attributes are grouped into four sections: structural, lexical, semantic, and code.

VI. 1. Structural Attributes

Structural attributes define the basic shape of the DIANA tree.

	9.1.B, 12.1.A]	
as_id_s:ID_S	[3.2.A, 3.2.B, 3.7.1, 5.1.B, 6.1.C, 7.4.B, 11.1]	34, 35, 42, 51, 58, 63, 70
as_item_s:ITEM_S	[5.6]	55
as_iteration:ITERATION	[5.5.A]	54
as_list:seq of ALTERNATIVE	[5.4]	53
as_list:seq of ARGUMENT	[App. I]	76
as_list:seq of CHOICE	[3.7.3.A]	43
as_list:seq of COMP	[3.7.A, 3.7.3.A]	41, 43
as_list:seq of COMP_ASSOC	[3.7.2, 4.3.A]	42, 47
as_list:seq of COMP_REP	[13.4.B]	75
as_list:seq of COMP_UNIT	[10.1.A]	69
as_list:seq of COND_CLAUSE	[5.3.A]	53
as_list:seq of CONTEXT_ELEM	[10.1.1.A]	69
as_list:seq of DECL	[7.1.B]	62
as_list:seq of DSCRMT_VAR	[3.7.1]	42
as_list:seq of DSCRT_RANGE	[3.6.A]	40
as_list:seq of ENUM_LITERAL	[3.5.1.A]	37
as_list:seq of EXP	[4.1.1]	46
as_list:seq of GENERIC_ASSOC	[12.3.A]	73
as_list:seq of GENERIC_PARAM	[12.1.B]	72
as_list:seq of ID	[3.2.C]	35
as_list:seq of ITEM	[3.9.B]	44
as_list:seq of NAME	[8.4, 9.10, 10.1.1.B]	64, 68, 70
as_list:seq of PARAM	[6.1.C]	58
as_list:seq of PARAM_ASSOC	[2.8.A]	33
as_list:seq of PRAGMA	[10.1.B]	69
as_list:seq of SELECT_CLAUSE	[9.7.1.A]	67
as_list:seq of STM	[5.1.A]	51
as_list:seq of VARIANT	[3.7.3.A]	43
as_membership_op:MEMBERSHIP_OP	[4.4.B]	46
as_name:NAME	[3.3.2.B, 3.6.B, 3.7.1, 3.7.3.A, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.6, 4.7, 5.2, 5.9, 6.1.C, 6.4, 7.4.B, 8.5, 9.5.B, 9.5.C, 10.2.A, 12.3.A, 13.3, 13.4.A, 13.4.B, 13.5, 13.8]	36, 40, 42, 43, 46, 47, 50, 52, 56, 59, 61, 63, 64, 66, 70, 73, 74, 75
as_name_s:NAME_S	[9.10]	68
as_name_void:NAME_VOID	[5.7, 6.1.B, 11.3]	56, 58, 71
as_object_def:OBJECT_DEF	[3.2.A, 3.7.1]	34, 42
as_package_def:PACKAGE_DEF	[7.1.A]	62
as_param_assoc_s:PARAM_ASSOC_S	[2.8.A, 6.4, 9.5.B]	33, 61, 66
as_param_s:PARAM_S	[6.1.B, 9.5.A, 9.5.C]	58, 66
as_pragma_s:PRAGMA_S	[10.1.B, 13.4.A]	69, 74
as_range:RANGE	[3.5.4, 13.4.B]	36, 75
as_range_void:RANGE_VOID	[3.5.7, 3.5.9]	39
as_record:INNER_RECORD	[3.7.3.A]	43
as_select_clause_s:SELECT_CLAUSE_S	[9.7.1.A]	67
as_stm:STM	[5.1.B, 5.5.A]	51, 54
as_stm_s1:STM_S	[9.7.2, 9.7.3]	68
as_stm_s2:STM_S	[9.7.2, 9.7.3]	68
as_stm_s:STM_S	[5.3.A, 5.4, 5.5.A, 5.6, 9.5.C, 9.7.1.A, 9.7.1.B]	53, 54, 55, 66, 67
as_subprogram_def:SUBPROGRAM_DEF 57		[6.1.A]
as_subunit_body:SUBUNIT_BODY	[10.2.A]	70
as_task_def:TASK_DEF	[9.1.A]	65
as_type_range:TYPE_RANGE	[4.4.B]	46
as_type_spec:TYPE_SPEC	[3.2.A, 3.3.1.A]	34, 35
as_unit_body:UNIT_BODY	[10.1.B]	69
as_variant_s:VARIANT_S	[3.7.3.A]	43

VI.2. Lexical Attributes

Lexical attributes represent information provided by the lexical analysis phase.

lx_comments:comments	[2.8.A, 3.2.A, 3.2.B, 3.2.C, 3.3.1.A, 3.3.2.A, 3.3.2.B, 3.4, 3.5, 3.5.1.A, 3.5.1.B, 3.5.4,	33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,

	3.5.7, 3.5.9, 3.6.A, 3.6.B, 3.7.A, 3.7.B, 3.7.1, 3.7.2, 3.7.3.A, 3.7.3.B, 3.8, 3.9.B, 4.1.A, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.3.A, 4.3.B, 4.4.A, 4.4.B, 4.4.D, 4.6, 4.7, 4.8, 5.1.A, 5.1.B, 5.1.F, 5.2, 5.3.A, 5.4, 5.5.A, 5.5.B, 5.6, 5.7, 5.8, 5.9, 6.1.A, 6.1.B, 6.1.C, 6.3, 6.4, 7.1.A, 7.1.B, 7.1.C, 7.4.A, 7.4.B, 8.4, 8.5, 9.1.A, 9.1.B, 9.5.A, 9.5.B, 9.5.C, 9.6, 9.7.1.A, 9.7.1.B, 9.10, 9.7.2, 9.7.3, 10.1.1.A, 10.1.A, 10.1.B, 10.1.1.B, 10.2.A, 10.2.B, 11.1, 11.3, 12.1.A, 12.1.B, 12.1.C, 12.1.D, 12.3.A, 13.3, 13.4.A, 13.4.B, 13.5, 13.8]	51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75
lx_default:Boolean	[6.1.C]	59
lx_numrep:number_rep	[4.4.D]	49
lx_prefix:Boolean	[6.4]	61
lx_srcpos:source_position	[2.8.A, 3.2.A, 3.2.B, 3.2.C, 3.3.1.A, 3.3.2.A, 3.3.2.B, 3.4, 3.5, 3.5.1.A, 3.5.1.B, 3.5.4, 3.5.7, 3.5.9, 3.6.A, 3.6.B, 3.7.A, 3.7.B, 3.7.1, 3.7.2, 3.7.3.A, 3.7.3.B, 3.8, 3.9.B, 4.1.A, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.3.A, 4.3.B, 4.4.A, 4.4.B, 4.4.D, 4.6, 4.7, 4.8, 5.1.A, 5.1.B, 5.1.F, 5.2, 5.3.A, 5.4, 5.5.A, 5.5.B, 5.6, 5.7, 5.8, 5.9, 6.1.A, 6.1.B, 6.1.C, 6.3, 6.4, 7.1.A, 7.1.B, 7.1.C, 7.4.A, 7.4.B, 8.4, 8.5, 9.1.A, 9.1.B, 9.5.A, 9.5.B, 9.5.C, 9.6, 9.7.1.A, 9.7.1.B, 9.10, 9.7.2, 9.7.3, 10.1.1.A, 10.1.A, 10.1.B, 10.1.1.B, 10.2.A, 10.2.B, 11.1, 11.3, 12.1.A, 12.1.B, 12.1.C, 12.1.D, 12.3.A, 13.3, 13.4.A, 13.4.B, 13.5, 13.8]	33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75
lx_symrep:symbol_rep	[3.2.A, 3.2.B, 3.3.1.A, 3.3.2.A, 3.5.1.B, 3.7.B, 3.7.1, 4.1.A, 4.4.D, 5.1.B, 5.5.A, 5.5.B, 6.1.A, 6.1.C, 7.1.A, 7.4.A, 9.1.B, 9.5.A, 11.1, 12.1.A, App. I]	34, 35, 36, 38, 41, 42, 45, 49, 51, 54, 55, 57, 59, 62, 63, 65, 66, 70, 71, 76

VI.3. Semantic Attributes

Semantic attributes represent the result of semantic analysis and provide information on the meaning of the program represented by the DIANA tree.

sm_actual_delta:Rational	[3.4, 3.5.9]	37, 39
sm_address:EXP_VOID	[3.2.A, 7.1.A, 9.1.A, 9.5.A]	34, 62, 65, 66
sm_base_type:TYPE_SPEC	[3.3.2.B, 3.5, 3.5.4, 3.5.7, 3.5.9]	36, 37, 38, 39
sm_bits:Integer	[3.5.9]	39
sm_body:BLOCK_STUB_VOID	[9.1.A, 9.1.B, 12.1.A]	65, 71
sm_body:PACK_BODY_DESC	[7.1.A]	62
sm_body:SUBP_BODY_DESC	[6.1.A]	57
sm_comp_spec:COMP_REP_VOID	[3.7.B, 3.7.1]	41, 42
sm_constraint:CONSTRAINT	[3.3.2.B, 4.1.2, 4.3.A, 4.4.D]	36, 46, 47, 49
sm_controlled:Boolean	[3.4, 3.8]	37, 44
sm_decl_s:DECL_S	[12.3.A]	73
sm_defn:DEF_OCCURRENCE	[4.1.A]	45
sm_discriminants:DSCRMT_VAR_S	[3.7.A, 7.4.A]	41, 63
sm_exception_def:EXCEPTION_DEF	[11.1]	70
sm_exp_type:TYPE_SPEC	[4.1.A, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.3.A, 4.4.A, 4.4.B, 4.4.D, 4.6, 4.7, 4.8, 6.4]	45, 46, 47, 48, 49, 50, 61
sm_first:DEF_OCCURRENCE	[3.2.A, 3.3.1.A, 3.7.1, 6.1.A, 6.1.C, 7.1.A, 9.1.B, 12.1.A]	34, 36, 42, 57, 59, 62, 65, 71
sm_generic_param_s:GENERIC_PARAM_S		

	[12.1.A]	71
sm_init_exp: EXP	[3.2.B]	35
sm_init_exp: EXP_VOID	[3.7.B, 3.7.1, 6.1.C]	41, 42, 58
sm_location: LOCATION	[6.1.A]	57
sm_normalized_comp_s: EXP_S	[3.7.2, 4.3.A]	42, 47
sm_normalized_param_s: EXP_S	[6.4, 9.5.B]	61, 66
sm_obj_def: OBJECT_DEF	[3.2.A]	34
sm_obj_type: TYPE_SPEC	[3.2.A, 3.2.B, 3.5.1.B, 3.7.B, 3.7.1, 5.5.B, 6.1.C]	34, 35, 38, 41, 42, 55, 58
sm_operator: operator	[4.1.A]	45
sm_packing: Boolean	[3.4, 3.6.A, 3.7.A]	37, 40, 41
sm_pos: integer	[3.5.1.B]	38
sm_record_spec: REP_VOID	[3.7.A]	41
sm_rep: integer	[3.5.1.B]	38
sm_size: EXP_VOID	[3.4, 3.5.1.A, 3.5.4, 3.5.7, 3.5.9, 3.6.A, 3.7.A, 3.8]	37, 38, 39, 40, [illegible]
sm_spec: GENERIC_HEADER	[12.1.A]	71
sm_spec: HEADER	[6.1.A, 9.5.A]	57, 66
sm_spec: PACKAGE_SPEC	[7.1.A]	62
sm_stm: LOOP	[5.7]	56
sm_stm: STM	[5.1.B, 5.5.A]	51, 54
sm_storage_size: EXP_VOID	[3.4, 3.8, 9.1.A]	37, 44, 65
sm_stub: DEF_OCCURRENCE	[6.1.A, 7.1.A, 9.1.B, 12.1.A]	57, 62, 65, 71
sm_type_spec: CONSTRAINED	[3.3.2.A]	36
sm_type_spec: TYPE_SPEC	[3.3.1.A, 7.4.A, 9.1.B]	35, 63, 65
sm_type_struct: TYPE_SPEC	[3.3.2.B, 3.5.4, 3.5.7]	36, 38, 39
sm_value: value	[4.1.A, 4.1.4, 4.4.A, 4.4.B, 4.4.D, 4.6, 4.7, 4.8, 6.4]	45, 47, 48, 49, 50, 61

VI.4. Code Attributes

Code attributes provide target-machine-specific information.

cd_impl_size: integer	[3.3.2.B, 3.4, 3.5.1.A, 3.5.4, 3.5.7, 3.5.9]	36, 37, 38, 39

REFERENCES

[1] P.F. Albrecht, P.E. Garrison, S.L. Graham, R.H. Hyerle, P. Ip, and B. Krieg-Bruckner.
Source-to-Source Translation: Ada to Pascal and Pascal to Ada.
In *Symposium on the Ada Programming Language*, pages 183-193. ACM-SIGPLAN, Boston, December, 1980.

[2] B. M. Brosgol, J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf.
$TCOL_{Ada}$: Revised Report on An Intermediate Representation for the Preliminary Ada Language.
Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department, February, 1980.

[3] J.N. Buxton.
Stoneman: Requirements for Ada Programming Support Environments.
Technical Report, DARPA, February, 1980.

[4] M. Dausmann, S. Drossopoulou, G. Goos, G. Persch, G. Winterstein.
AIDA Introduction and User Manual.
Technical Report Nr. 38/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.

[5] M. Dausmann, S. Drossopoulou, G. Persch, G. Winterstein.
On Reusing Units of Other Program Libraries.
Technical Report Nr. 31/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.

[6] *Formal Definition of the Ada Programming Language*
November 1980 edition, Honeywell, Inc., Cii Honeywell Bull, INRIA, 1980.

[7] J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, H.F. Ledgard, J.C. Heliard, J.R. Abrial, J.G.P. Barnes, M. Woodger, O. Roubine, P.N. Hilfinger, R. Firth.
Reference Manual for the Ada Programming Language
The revised reference manual, July 1980 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1980.

[8] J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, H.F. Ledgard, J.C. Heliard, J.R. Abrial, J.G.P. Barnes, M. Woodger, O. Roubine, P.N. Hilfinger, R. Firth.
Reference Manual for the Ada Programming Language
Draft revised MIL-STD 1815, July 1982 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1982.

[9] J.R. Nestor, W.A. Wulf, D.A. Lamb.
IDL - Interface Description Language: Formal Description.
Technical Report CMU-CS-81-139, Carnegie-Mellon University, Computer Science Department, August, 1981.

[10] G. Persch, G. Winterstein, M. Dausmann, S. Drossopoulou, G. Goos.
AIDA Reference Manual.
Technical Report Nr. 39/80, Institut fuer Informatik II, Universitaet Karlsruhe, November, 1980.

[11] Author unknown.
Found on a blackboard at Eglin Air Force Base.

INDEX

Diana Classes

Diana Nodes

Diana Attributes

– 8

DTIC